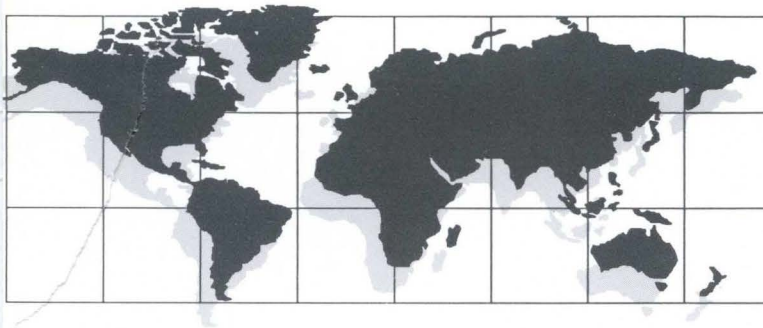


CTOS

Programming Guide

Volume 2
Extended System Services and Libraries

UNISYS

UNISYS

CTOS®

Programming Guide

Volume 2
Extended System Services and Libraries

Copyright © 1990, 1991, 1992, 1993 Unisys Corporation
All Rights Reserved
Unisys is a registered trademark of Unisys Corporation

CTOS III 1.1
CTOS II 3.4
CTOS I 3.4
CTOS/XE 3.4.100
Development Utilities 12.3
Standard Software 12.3
Video Access Method 4.2

November 1993

Printed in USA
4586 7918-000

The names, places, and/or events used in this publication are not intended to correspond to any individual, group, or association existing, living, or otherwise. Any similarity or likeness of the names, places, and/or events with the names of any individual, living or otherwise, or that of any group or association is purely coincidental and unintentional.

NO WARRANTIES OF ANY NATURE ARE EXTENDED BY THIS DOCUMENT. Any product or related information described herein is only furnished pursuant and subject to the terms and conditions of duly executed agreement to purchase or lease equipment or to lease software. The only warranties made by Unisys, if any, with respect to the products described in this document are set forth in such agreement. Unisys cannot accept any financial or other responsibility that may be the result of your use of the information in this document or software material, including direct, special or consequential damages.

You should be careful to ensure that the use of this information and/or software material complies with the laws, rules, and regulations of the jurisdictions with respect to which it is used.

The information contained herein is subject to change without notice. Revisions may be issued to advise of such changes and/or additions.

RESTRICTED RIGHTS LEGEND. Use, reproduction, or disclosure is subject to the restrictions set forth in DFARS 252.227-7013 and FAR 52.227-14 for commercial computer software.

Copyright © 1993 Unisys Corporation
All Rights Reserved
Unisys is a registered trademark of Unisys Corporation

Convergent, Convergent Technologies, CTOS, NGEN, and SuperGen are registered trademarks of Convergent Technologies, Inc.

ClusterCard, Context Manager, CT-Net, CTOS/VM, CTOS/Vpc, Document Designer, Generic Print System, PC Emulator, Print Manager, shared resource processor, SRP, Voice/Data Services, Voice Processor, X-Bus, and X-Bus+ are trademarks of Convergent Technologies, Inc.

OFIS is a registered trademark of Unisys Corporation.

BTOS is a trademark of Unisys Corporation.

OS/2 is a registered trademarks of International Business Machines Corporation. Microsoft is a registered trademark of Microsoft Corporation. Presentation Manager and Windows are trademarks of Microsoft Corporation.

UNISYS

Product Information Announcement

● New Release ○ Revision ○ Update ○ New Mail Code

Title

CTOS® Programming Guide, Volume 2, Extended System Services and Libraries

This Product Information Announcement announces the release and availability of the CTOS Programming Guide, Volume 2, Extended System Services and Libraries.

This two-volume set describes general programming techniques and the extended system services provided with the CTOS operating systems. It is meant to be used in conjunction with *CTOS/Open Programming Practices and Standards.* This new edition includes information on programming Standard VGA, writing dynamic link libraries under CTOS, and writing system services for multiprocessor EISA-bus workstations.

Please address all technical communications relative to this document to UNISYS, Multimedia Publishing M/S 18-007, 2700 North First Street, San Jose, CA 95134-2028.

To order additional copies of this document

- United States customers, call Unisys Direct at 1-800-448-1424.
- All other customers, contact your Unisys Sales Office.
- Unisys personnel, use the Electronic Literature Ordering (ELO) system.

Contact your Unisys representative for pricing information.

4586 7900-000 Volume 1 only
4586 7918-000 Volume 2 only
4357 9515-100 Volumes 1 and 2

CTOS is a registered trademark of Convergent Technologies, Inc.

Announcement only:
SA, SN, SU, SW, PR5

Announcement and attachments:

System: CTOS
Release:
CTOS III 1.1
CTOS II 3.4
CTOS I 3.4
CTOS/XE 3.4.100
Development Utilities 12.3
Standard Software 12.3
Video Access Method 4.2

Part Number: 4586 7918-000

Contents

Volume 1

About This Guide

Introduction xix
What This Guide Covers xix
Who Should Use This Guide xx
How This Guide Is Organized xx
What Is New in This Guide xxii
Where to Find More Information xxiv

Section 1. The CTOS Programming Environment

Introduction to CTOS Programming 1-1
Distributed Client-Server Computing: The CTOS Cluster 1-1
Tools Available to the CTOS Programmer 1-2
CTOS Development Utilities 1-2
Language Compilers and Interpreters 1-2
Additional Programmer's Libraries 1-3
An Overview of CTOS System Software 1-3
General Steps in Writing a CTOS Program 1-6
Writing Source Code 1-6
Compiling a CTOS Program 1-7
Linking a CTOS Program 1-7
Running the Program 1-8
Naming Conventions 1-9
Data Types 1-9
Defining the Basic Data Types 1-10
Variable Names 1-10
Prefixes 1-10
Roots 1-12
Suffixes 1-13
Examples of Variable Names 1-13
Procedure Naming 1-14
File Suffix Conventions 1-14
Models of Segmentation 1-17
Your Compiler and Models of Segmentation 1-17
Procedural Calls Between Programming Languages 1-18
Your Compiler and Calls Between Languages 1-18
Some Configuration Tips for Programmers 1-19
The CTOS Debugger 1-19
Using Consistent Library Versions 1-20

Section 2 Introduction to Protected Mode

Introduction to Protected Mode 2-1
Advantages of Protected Mode 2-2
Review of Segmented Addressing 2-2
Review of Real-Mode Addressing 2-3
Protected Mode Addressing 2-4
Memory Mapping Using 80386 Paging 2-6
The Format of a Selector 2-7
The Segment Descriptor Tables 2-7
Local Descriptor Table (LDT) 2-7
Global Descriptor Table (GDT) 2-8
The Contents of a Segment Descriptor 2-9
Processor Exceptions and Faults 2-11
Gate Descriptors 2-12
Protection Models 2-12

Section 3 Using Color

Introduction to Color 3-1
Video Concepts 3-2
Program Interfaces 3-4
ProgramColorMapper 3-5
ProgramColorMapper Structures 3-6
Single-Palette Format 3-7
Advantages Over Three-Palette Format 3-9
Three-Palette Format 3-9
Color Structure 3-11
Field Descriptions 3-12
Alphanumeric, Background, and Graphics 3-14
Enabling Background Color Capability 3-14
Effect of fBackgroundColor on Palettes 3-15
Background Color and Three-Palette Format 3-15
Background Color and Single-Palette Format 3-16
More Examples in Three-Palette Format 3-18
Case 1: Using Alphanumeric Control Structure Fields . 3-18
Case 1: Condition A 3-18
Case 1: Condition B 3-19
Case 1: Condition C 3-20
Case 1: Condition D 3-20
Case 2: Using Graphics Control Structure Fields 3-21
Case 2: Condition A 3-21
Case 2: Condition B 3-22
Case 3: Using Alpha and Graphics Fields 3-22

Maximizing Use of a 16-Entry Palette 3-23
Gray-Scale Monitors 3-23
Color Application Notes 3-24
Combining Alphanumeric Color With the Values in the Attribute Byte 3-24
Defining Bitmap Color 3-27
Color Priorities 3-28
Color Programming Tips 3-30
Avoid Use of Reverse Video and Graphics 3-30
Querying Single-Palette Format 3-30
Avoid Combining Alpha Background with Graphics 3-31
Color Program Examples 3-31
Three-Palette Format Example 3-32
Single-Palette Format Example 3-40

Section 4 Converting CTOS VGA to Standard VGA

Overview 4-1
Who Should Use This Information 4-2
General Concepts and Terminology 4-2
VGA Font Capabilities 4-6
Nine-Bit-Wide Fonts 4-6
Standard VGA Capability 4-7
CTOS VGA Capability 4-8
Problem Description 4-10
Affected Applications 4-10
Recommended Solution 4-10
Half-Bit Shift 4-12
Standard VGA Capability 4-12
CTOS VGA Capability 4-12
Problem Description 4-13
Affected Applications 4-13
Recommended Solution 4-13
Display Resolution 4-15
Standard VGA Capability 4-15
CTOS VGA Capability 4-16
Problem Description 4-16
Affected Applications 4-17
Recommended Solution 4-17

Character Attributes 4-17
Standard VGA Capability 4-17
CTOS VGA Capability 4-20
Problem Description 4-21
Affected Applications 4-22
Attributes 4-22
Color 4-22
Character Map 4-22
Recommended Solution 4-23
Mapping Attributes from CTOS to Standard VGA 4-23
Default Attribute Mapping 4-24
Default Attribute Mapping Tables 4-29
Accessing the Character Map 4-40

Section 5 Writing Partition-Managing Programs

Introduction to Partition Management 5-1
A Review of Partition Management Operations 5-1
Creating a Partition and Loading a Program Into It 5-2
Creating a Partition 5-2
Swapping the Partition Context Into Memory 5-3
Setting Up the Program's Environment 5-4
Loading the Program into the New Partition 5-6
Finding the Program's Termination Status 5-6
The Child-Termination Question 5-6
A Solution: Defining a Termination Request 5-7
Calling the Termination Procedure 5-7
Structure of the Termination Procedure 5-8
Deallocating the Partition 5-9
A Sample Partition-Management Program 5-10

Section 6 Software Installation: The Installation Manager

Introduction to Installation Manager 6-1
Key Concepts 6-2
Types of Installation: Floppy, Tape, Server 6-2
Public vs. Private Installation 6-3
Subpackages 6-3
The Installation Manager 6-4
Batch 6-6
Installation Files 6-6
Control File 6-7
Control File Keywords 6-8
Installation Script File 6-13
Message File 6-13
Command File 6-14
User Configuration File 6-15
Naming Your Floppy Installation Files 6-16
Naming Your Tape Installation Files 6-18
Organizing Your Installation Media 6-20
For a Floppy Installation 6-20
For a Tape Installation 6-23
Installation Variables 6-23
File Lists 6-27
Restarting an Installation 6-28
Nationalization 6-29
Installation Tips 6-30
Installation Manager Version 12.0 and Earlier 6-30
Using Variables 6-31
Messages 6-32
Packages and Subpackages 6-32
Volumes and Directories 6-32
File Lists 6-33
Other Tips 6-33
Installation Script Examples 6-34
Example 1: One Subpackage 6-35
Install.ctrl File 6-35
InstallMsg.bin File 6-35
Install.cmds File 6-35
Install.jcl File 6-36

Example 2: Nested Subpackages 6-41
Install.ctrl File 6-42
DevelopmentUtilities>Install.ctrl File 6-42
DevelopmentRunFiles>Install.ctrl File 6-43
DevelopmentRunFiles>InstallMsg.bin File 6-43
DevelopmentRunFiles>Install.cmds File 6-43
DevelopmentRunFiles>Install.jcl File 6-44
DevelopmentLibraries>Install.ctrl File 6-49
DevelopmentLibraries>InstallMsg.bin File 6-50
DevelopmentLibraries>Install.jcl File 6-50
AsynchronousExamples>Install.ctrl File 6-55
AsynchronousExamples>InstallMsg.bin File 6-55
AsynchronousExamples>Install.cmds File 6-55
AsynchronousExamples>Install.jcl File 6-56
Example 3: Tape Installation 6-63
InstallMsg.bin File 6-64
Install.jcl File 6-64

Section 7 Using the System Log File

Introduction to System Log File 7-1
Log File Format 7-2
Writing Messages to the Log File 7-2
Displaying Messages Using PLog 7-3
Log File Fields in the Volume Home Block 7-3
How the File System Writes to the Log File 7-4
Record Fits in the Buffer 7-4
Record Does Not Fit in the Buffer 7-4
How to Read the Log File 7-6
Accessing the Log-File Fields in the VHB 7-6
Determining if Records Have Wrapped Around 7-6
Reading the File in Chronological Order 7-7
If Records Have Not Wrapped Around 7-7
If Records Have Wrapped Around 7-8
Reading the File in Reverse Chronological Order 7-8
Saving Offsets to Records 7-8
If Records Have Not Wrapped Around 7-9
If Records Have Wrapped Around 7-9
PLog's Algorithm For Processing Each Sector ... 7-10

Section 8 Writing Multiprocessor System Services

Introduction 8-1
Portation Issues for Existing Programs 8-1
General Guidelines 8-2
Unsupported Mechanisms 8-2
User Numbers and Exchanges 8-2
Remote Memory and Inter-CPU Communication 8-3
Restricted Use of GetWSUserName on SRPs 8-3
Controlling Interprocessor Request Routing 8-3
Interprocessor Request Routing Types 8-4
Routing by Device Specification 8-6
Deinstalling an Instance of a System Service 8-14
Remote Routing 8-15
Local Routing 8-18
Other Routing Methods 8-19
Enhancing Remote Data Transfer Requests on SRPs .. 8-19
Use of Handles on Multiprocessor Systems 8-20
The Standard Connection Handle 8-21
Non-Standard Handle Types 8-21

Section 9 The Synchronous CommLine Interface

Introduction to CommLine Interface 9-1
The CommLine Interface 9-1
Extensions to the Traditional CommLine Interface 9-1
InitCommLine 9-2
Differences Between 8274 and 8530 Communication Controllers 9-3
ChangeCommLineBaudRate 9-4
ReadCommLineStatus 9-4
WriteCommLineStatus 9-5
Using DMA with Synchronous Data Communication 9-5
Initializing Communications DMA 9-6
TransmitCommLineDMA 9-7
ReceiveCommLineDMA 9-7
GetCommLineDMAStatus 9-8
Implementing X.21 (1984) Protocol Support 9-9
What Is the X.21 Protocol? 9-9
Features of the X.21 Support Hardware 9-10
Initializing a Communications Line with X.21 Support Enabled 9-12
Using the X.21 Support Hardware in Drivers-Only Mode 9-12
Using V.35 Support Hardware 9-13

Section 10 Creating a Dynamic Link Library

Introduction 10-1
Assumptions 10-1
Supporting Documentation 10-2
Software and Tools 10-2
Introducing the DLL Example 10-4
Queue DLL 10-4
DLL Source Files 10-5
Overview of Queue.c 10-6
Queue.c Declarations 10-6
Global Data Declarations 10-6
Instance Data Declarations 10-6
Queue Pointer Procedures 10-7
IncrementQueuePtr 10-7
QueueUsedSpace 10-8
QueueFreeSpace 10-8
ReadQueueIntoMemory 10-9
WriteQueueFromMemory 10-9
Initialization and Termination Procedures 10-9
Export Procedures 10-10
Mediating Medium-Model Library Procedures 10-10
Setting Up the Initialization Procedure 10-11
Using a Termination Procedure 10-12
Using Semaphores to Handle Global Data 10-14
In QueueInitialization 10-14
In QueueTermination 10-15
In The Export Procedures 10-16
Setting Up Global Data Variables 10-17
Setting Up Instance Data Variables 10-18

Preparing the DLL 10-19
Overview of Preparing a DLL 10-20
Makefile and Executive Commands 10-22
Compiling Queue.c 10-23
Creating the Object Module and Import Library .. 10-24
Linking The DLL Object and Library Files 10-25
Using the DLL 10-27
Client Source Files 10-27
Reading Queue Entries 10-28
Writing Queue Entries 10-28
Preparing the Client Source Files 10-29
Client Makefile and Executive Commands 10-31
Compiling ReadQueue.c 10-31
Linking the Client Object and Library Files 10-32
Adding Resources 10-34
DLL Program Listings 10-35

Volume 2

Section 11 Mouse Services

What Are the Mouse Services? 11-1
Mouse System Service 11-1
Object Module Library 11-1
Functional Groups 11-2
Examples 11-3
Important Concepts 11-3
Using the Mouse Buttons 11-4
Screen Coordinates 11-4
Normalized Screen Coordinates 11-4
Virtual Screen Coordinates 11-4
Setup 11-6
Queries 11-7
Cursor Shape 11-12
Changing the Graphics Cursor 11-14
Cursor Movement 11-15
Troubleshooting 11-16
Sample Mouse Program 11-17

Section 12 Queue Manager

Introduction to Queue Manager 12-1
What Is the Queue Manager? 12-1
Run Files 12-2
Installation and Deinstallation 12-2
Functional Groups of Queue Management Operations ... 12-3
Queue Manipulation Operations 12-3
Queue Server Operations 12-4
Establishing Queue Servers 12-4
Marking Queue Entries 12-4
Unmarking Queue Entries 12-5
Processing Queue Entries 12-5
Removing and Rescheduling Queue Entries 12-5
Rewriting Queue Entries 12-5
Client Operations 12-6
Adding an Entry to a Queue 12-6
Deleting an Entry 12-7
Reading Queue Entries 12-7
Summary of Queue Manager Operations 12-7
Queue Manager Configuration 12-10
Queues 12-11
Format 12-11
Queue File Header 12-11
Queue Entry 12-12
Queue Entry Processing Order 12-12
Queue Entry Format 12-13
Calculating Queue Entry Size 12-13
Queue Examples 12-14
Defining Queues 12-14
Defining Queues in the Queue Index File 12-14
Defining Queues Dynamically 12-16
Referencing Queues and Queue Entries 12-17
Referencing Queues 12-17
Queue Names 12-17
Queue Handles 12-18
Referencing Queue Entries 12-18
Queue Entry Handles 12-18
Queue Status Block 12-19
Keys 12-19
Sequence for Using Queue Management Operations 12-21
Using the Queue Manager Across Network Nodes 12-22
Data Structures 12-24

Section 13 Spooler

Introduction to Spooler 13-1
Spooler Configuration 13-1
Sending a Password 13-2
Operations 13-2
Programmer's Notes on the Spooler 13-3
Pre-GPS Spooler Byte Streams 13-3
Spooler Configuration File Requirements 13-4
Printer Spooler Escape Sequences 13-5
Queue Management 13-5
Scheduling Queue 13-6
Status Queue 13-6
Control Queue 13-6
ConfigureSpooler Program Example 13-7
SpoolerPassword Program Example 13-7
Data Structures 13-8

Section 14 Voice/Data Services

Overview 14-1
Telephone Management 14-2
Telephone Status Monitor Program 14-2
Data Management 14-2
Functional Groups of Operations 14-2
Voice Management Operations 14-3
Telephone Management Operations 14-3
Data Management Operations 14-5
Hardware Features of the X-Bus Workstation 14-6
Voice Features 14-6
CODEC 14-6
Voice Amplifier 14-6
Telephone Features 14-6
Two Telephone Lines 14-7
DTMF Tone Generator and Receiver 14-7
Call Progress Tone Detector (CPTD) 14-9
Data Features 14-9
Modem 14-9
Analog Crosspoint Switch Array 14-9

Software Concepts 14-10
Voice Recording 14-10
Recording Rates 14-10
Pause Compression 14-10
Amplification 14-11
Memory and Disk Files 14-12
Memory Usage 14-12
Structure of a Voice File 14-13
Adaptive Differential Pulse Code Modulation (ADPCM) 14-13
Voice Control Structure 14-14
Typical Sequence for Record/Playback 14-14
Multiple Voice Messages in One File 14-14
Voice Playback from Memory 14-15
Telephony 14-15
Voice and Data Lines 14-15
Telephone Unit vs. Telephone Line 14-16
Hold 14-16
Dialing 14-16
Generating DTMF Tones 14-18
Internationalized Call Progress Tone Detection .. 14-18
Sample Program Using Internationalized Call Progress-Tone Detection System 14-21
Data 14-22
Starting a Data Call 14-23
Converting a Voice Call to a Data Call 14-23
Accepting a Data Call 14-23
Originating a Data Call 14-24
Reading and Writing Data 14-24
Terminating a Data Call 14-24
Telephone Status Debugging Tool 14-25
Command Form 14-25
Parameter Field 14-25
Operation 14-26
Lines J0 through J3 14-27
Lines L0 through L7 14-28
Status Monitor Function Keys 14-29

Program Examples 14-30
Example 14-2: Dialing 14-30
Example 14-3: Voice Response System 14-35
Setting Up Request Blocks 14-35
fNoWait Flag 14-35
Use of TsVoiceConnect 14-36
Possible Modifications to This Program 14-36
Example 14-4: Voice Memory Playback 14-57
Example 14-5: Data Call 14-69
Data Structures 14-81

Section 15 Performance Statistics Service

Overview 15-1
Functional Groups of Operations 15-2
Statistics Session Operations 15-2
Logging Session Operations 15-2
General-Purpose Operations 15-2
Statistics Session 15-3
Statistics ID Block 15-3
Block Number 15-3
Index 15-3
Opening a Statistics Session (PsOpenStatSession) ... 15-4
Block ID 15-4
Session Handle 15-5
Disk Activity 15-5
Getting the Counters (PsGetCounters) 15-6
Closing a Statistics Session (PsCloseSession) 15-8
Logging Session 15-8
Opening a Logging Session (PsOpenLogSession) 15-8
Reading a Log (PsReadLog) 15-9
Closing a Logging Session (PsCloseSession) 15-10
Program Example 15-10
Data Structure 15-17

Section 16 Asynchronous System Service Model

Introduction to Asynchronous Services 16-1
Terminology 16-1
Synchronous and Asynchronous System Service Models 16-3
Writing an Asynchronous System Service 16-6

Async.lib Procedures 16-10
Async.lib Procedures You Can Use in the Main Module .. 16-10
Requesting a Service on Behalf of a Client 16-10
Review of the Synchronous Request Procedural Interface 16-10
Asynchronous Request Procedural Interface 16-11
Passing a Variable Length Parameter List in PL/M 16-12
Building Request Blocks 16-14
AsyncRequest 16-14
AsyncRequestDirect 16-14
Checking the Context Stack 16-15
CheckContextStack 16-15
Heap 16-15
Allocating and Deallocating Heap Memory 16-16
Conserving Heap Memory 16-17
System Requests 16-17
Terminating and Aborting Requests 16-17
Swapping Requests 16-18
Handling System Requests 16-19
Handling Termination and Abort Requests 16-19
Handling Swapping Requests 16-21
Debugging Aids 16-22
Examining the Logging Module LogAsync 16-22
Maintaining Debugging Statistics 16-24
Async.lib Procedures Used by the Common-Code Module 16-25
Managing Contexts 16-25
Creating a Context 16-26
Context Control Block 16-27
Resuming a Context 16-27
Terminating a Context 16-28
Other Ways Contexts Can Be Used 16-28
Terminating Contexts at Deinstallation 16-29
Using the Heap 16-29
Managing the Heap 16-30
Logging Messages for Debugging Purposes 16-32
Initializing 16-32
Availability of Asynchronous System Service Files 16-33
Linking Your System Service 16-34
Program Example 16-36
AsyncService.c 16-36
Example.c 16-37
Start.c, Stop.c, and DeInstall.c 16-37

Section 17 CD-ROM Service

CD-ROM Overview 17-1
Requirements 17-1
Functional Groups of Operations 17-2
Volume Information 17-2
Status 17-3
Read File 17-3
Audio 17-3
Miscellaneous (Rarely Used) 17-4
Standard File Formats 17-4
Determining the File Format Using CdGetVolumeInfo .. 17-4
Other Uses of CdGetVolumeInfo 17-6
File Structure: Hierarchical vs. Flat 17-6
CTOS File Specification (for Flat File Structures)..... 17-7
Backslash File Specification (for Hierarchical File Structures) 17-8
Obtaining the Directory List 17-9
Example: Using CdDirectoryList 17-10
Searching for Files 17-13
Example: File Search 17-14
Copying a CD-ROM File to Disk 17-19
Example: Copying to Disk 17-19
Using Audio Features of CD-ROM 17-24
Specifying Locations on the CD-ROM Disc 17-24
Q-Channel 17-24
Audio Example 17-25
File Formats 17-33
Primary Volume Descriptor Formats 17-33
ISO Directory Record Format 17-49
Character Sets 17-58
d-characters 17-58
a-characters 17-59
c-characters 17-60
a1-characters 17-60
d1-characters 17-60
Separators 17-60

Section 18 Sequential Access Service

Overview of Sequential Access Service 18-1
Functional Groups of Operations 18-2
Basic Operations 18-2
Advanced Operations 18-3
Miscellaneous 18-3
General Model of Sequential Access Devices 18-4
Data Storage Characteristics 18-4
Logical Elements within a Tape 18-6
Logical Data Blocks 18-6
Interblock Gaps 18-7
Filemarks 18-7
Blank Space 18-7
Erase Gaps 18-7
Data Buffering 18-8
Device Buffers 18-8
Sequential Access Service Buffers 18-9
Allocating Memory for Recovering Buffer Data 18-9
Residual Data 18-9
Buffer Recovery Order 18-12
Example 18-13
SeqAccessCheckpoint 18-14
Specifying Buffer Sizes 18-16
Fixed-Length and Variable-Length Records 18-16
Increasing Block Size: Pros and Cons 18-17
Programming Considerations 18-17
Record Size and Block Size 18-17
Recording Density and Transport Speed 18-18
Buffered Mode 18-19
Erase to EOM After Close 18-19
Suppress Default Mode on Open 18-19
Buffer Recovery Order 18-19
Examples 18-20
Example 1: Fixed-Length, Blocked Records 18-20
Example 2: Variable-Length Records 18-24

Figures

1-1. Where CTOS System Calls Are Processed 1-5

2-1. Real Mode Segment Addressing 2-3
2-2. Protected Mode Segment Addressing 2-5
2-3. Protected Mode Address Translation for Paging 2-6
2-4. The Format of a Selector 2-7
2-5. Pointer Aliasing 2-10
2-6. Separate Address Spaces Protection Model 2-13

3-1. Character Cell with Foreground and Background Pixels 3-3
3-2. Example Palette in the Single-Palette Format 3-8
3-3. Three-Palette Format and Single-Palette Format 3-13
3-4. Sample Three-Palette Format 3-15
3-5. Sample Single-Palette Format 3-17
3-6. Case 1 Palette Settings 3-18
3-7. Case 2 Palette Settings 3-21
3-8. Sample Palette 3-28
3-9. Color Priorities 3-29

4-1. Character Cell 4-3
4-2. Scan Lines 4-4
4-3. Relationship of a Character Cell to a Screen Display 4-5
4-4. Characters Codes 0 through 127 4-7
4-5. Characters Codes 128 through 255 4-8
4-6. Representing 9-Bit-Wide Characters 4-9
4-7. Converting to an 8x12 Font 4-11
4-8. Half-Bit Shift 4-12
4-9. Effects of Ignoring Half-Bit Shift 4-14
4-10. Standard VGA: Option One 4-17
4-11. Standard VGA: Option Two 4-18
4-12. Standard VGA: Option Three 4-18
4-13. CTOS Attributes Without Color 4-20
4-14. CTOS Attributes With Alphanumeric Color 4-20
4-15. Standard VGA Default Attributes 4-24
4-16. Mapping CTOS Attributes 4-41

6-1 Files for Floppy Installation of Accounting Package 6-17
6-2 Files Used in a Tape Installation for an Accounting Package 6-19

7-1. Log File Wraparound 7-5

10-1 Queue DLL 10-4
10-2. IncrementQueuePtr 10-8
10-3. Leaving-One-Byte-Free 10-9
10-4. Preparing the DLL 10-21
10-5. Preparing the ReadQueue Client 10-30

11-1. Virtual and Normalized Screen Coordinates 11-5
11-2. Cursor Movement Inside and Outside a Motion Rectangle 11-9

12-1. Configuration with the Queue Management Facility 12-10
12-2. Example of a Queue Index File 12-16

14-1. Parts of the Telephone Unit 14-7
14-2. Voice Processor Module Connections 14-8
14-3. Telephone Status Command Screen 14-26
14-4. Telephone Status Screen with Telephone Offhook 14-27

16-1. Program Flow for the Synchronous Model 16-5
16-2. Program Flow for the Asynchronous Model 16-7
16-3. Source Modules to Run File 16-9

17-1. Flat File Structure 17-7
17-2. Hierarchical File Structure 17-8

18-1. General Layout of a Tape 18-5
18-2. Serpentine Recording (QIC Tape) 18-5
18-3. Parallel Recording (Half-inch Tape) 18-6
18-4. Helical Scan Recording (DDS) 18-6
18-5. Example of Variable-Length Records 18-18

Tables

1-1. Basic Data Types 1-9
1-2. Common Prefixes 1-11
1-3. Common Roots 1-12
1-4. Common File Suffixes 1-15

4-1. Display Resolutions Supported by CTOS Standard VGA VAM 4-15
4-2. CTOS VGA Display Resolutions 4-16
4-3. Standard VGA Default Colors 4-19
4-4. CTOS VGA Default Colors 4-21
4-5. CTOS Standard VGA VAM Default Colors 4-25
4-6. Standard VGA Attributes (0 - F) 4-26
4-7. Standard VGA Attributes (10 - 1F) 4-27
4-8. Standard VGA Attributes (x0 - xF) (where x = (2,3,4,5,6,7)) 4-28
4-9. Standard VGA Attributes (x0 - xF) (where x = (8,9,A,B,C,D,E,F)) ... 4-29
4-10. Monochrome Attribute Mapping (00 - 0F) 4-30
4-11. Monochrome Attribute Mapping (10 - 1F) 4-30
4-12. Monochrome Attribute Mapping (20 - 2F) 4-31
4-13. Monochrome Attribute Mapping (30 - 3F) 4-31
4-14. Color Attribute Mapping (00 - 0F) 4-32
4-15. Color Attribute Mapping (10 - 1F) 4-32
4-16. Color Attribute Mapping (20 - 2F) 4-33
4-17. Color Attribute Mapping (30 - 3F) 4-33
4-18. Color Attribute Mapping (40 - 4F) 4-34
4-19. Color Attribute Mapping (50 - 5F) 4-34
4-20. Color Attribute Mapping (60 - 6F) 4-35
4-21. Color Attribute Mapping (70 - 7F) 4-35
4-22. Color Attribute Mapping (80 - 8F) 4-36
4-23. Color Attribute Mapping (90 - 9F) 4-36
4-24. Color Attribute Mapping (A0 - AF) 4-37
4-25. Color Attribute Mapping (B0 - BF) 4-37
4-26. Color Attribute Mapping (C0 - CF) 4-38
4-27. Color Attribute Mapping (D0 - DF) 4-38
4-28. Color Attribute Mapping (E0 - EF) 4-39
4-29. Color Attribute Mapping (F0 - FF) 4-39
4-30. Operations for Updating the Screen Indirectly 4-42
4-31. Operations for Updating the Screen Directly 4-43

8-1. Interprocessor Request Routing Types 8-4

9-1. 8274 and 8530 Register Differences 9-3
9-2. X.21 Status Indication Bit Patterns 9-10

11-1. Mouse Procedures by Function 11-2

2-1. Queue Examples 12-14
12-2. Queue File Header 12-24
12-3. Queue Entry Header 12-27
12-4. Queue Status Block 12-29

13-1. Spooler Scheduling Queue Entry 13-8
13-2. Spooler Status Queue Entry 13-11
13-3. Spooler Control Queue Entry 13-14

14-1. Dial Characters 14-17
14-2. Telephone Service Configuration File Format 14-81
14-3. Telephone Service Configuration Block 14-82
14-4. Telephone Status Structure 14-85
14-5. tUnitState Fields of the Telephone Status Structure 14-87
14-6. lineState Fields of the Telephone Status Structure 14-89
14-7. codecState Fields of theTelephone Status Structure 14-92
14-8. Voice File Header 14-93
14-9. rgMessages Fields of theVoice File Header 14-94
14-10. Voice File Record 14-96
14-11. Voice Control Structure 14-97
14-12. Data Control Structure 14-100

15-1. Performance Statistics Structure 15-17

17-1. ISO Primary Volume Descriptor 17-33
17-2. High Sierra Primary Volume Descriptor 17-42
17-3. ISO Directory Record Format 17-49
17-4. High Sierra Directory Record Format 17-54

Examples

2-1. Special Example 2-9

3-1. Another Example in the Next Chapter 3-4
3-1. ThreePalette.c 3-32
3-2. NewPalette.c 3-40

5-1. Creating a Partition 5-2
5-2. Initializing a New Partition's Environment 5-4
5-3. Loading a Program for Execution 5-6
5-4. Calling a Child-Termination Procedure 5-7
5-5. Child-Termination Procedure 5-8
5-6. Partition Removal Procedure 5-9
5-7. Simple Partition-Managing Program 5-10
5-8. Termination Request Definiton File 5-18

7-1. PLog's Record-Processing Algorithm 7-10

8-1. Sample Request.txt File Using Device Routing 8-7
8-2. Client Program that Supports Device Routing 8-10
8-3. Changes to Illustrate Device Routing 8-12
8-4. Routed Deinstallation Request 8-14
8-5. Sample Request.txt File Using Remote Routing 8-16

10-1. Queue.c 10-35
10-2. Queue.def 10-46
10-3. Dll.mak 10-47
10-4. Queue.h 10-49
10-5. ReadQueue.c 10-50
10-6. WriteQueue.c 10-51
10-7. Clients.mak 10-55
10-8. LinkerConfig.sys 10-57

Section 11
Mouse Services

What Are the Mouse Services?

The Mouse Services are a set of commands and software programs for interfacing the mouse with the operating system and application programs.

Mouse System Service

The Mouse System Service is a system service that contains the programming request and procedural interfaces for the mouse. It handles cursor control and tracking, the main activities of the mouse software. Application programmers will use the Mouse System Service and the object module library described below.

To use the Mouse Services, first install the Mouse System Service on your workstation, as described in the *CTOS Executive Reference Manual* and the *CTOS System Administration Guide.*

Note: *Although the procedures described here support any pointing device, this manual uses the term mouse for the pointing device.*

Object Module Library

The object module library provides high-level procedures that deal with the mouse, such as coordinate translations from one coordinate system to another.

Functional Groups

Table 11-1 groups the mouse procedures according to function. The examples that follow provide further explanation of procedures dealing with setup, queries, cursor shape, and cursor movement. See the *CTOS Procedural Interface Reference Manual* for complete descriptions of these procedures.

Table 11-1. Mouse Procedures by Function

Setup	Queries
PDSetCursorType	GetIBusDevInfo
PDSetTracking	PDGetCursorPos
PDSetVirtualCoordinates	PDGetCursorPosNSC
PDSetMotionRectangle	PDQueryControls
PDSetMotionRectangleNSC	PDQuerySystemControls
PDInitialize	ReadInputEvent
	ReadInputEventNSC
Cursor Shape	**Cursor Movement**
PDLoadCursor	PDSetCursorPos
PDLoadSystemCursor	PDSetCursorPosNSC
PDReadCurrentCursor	PDSetCursorDisplay
PDReadIconFile	
Controls	**Transformations**
PDSetControls	PDTranslateVCtoNSC
PDSetSystemControls	PDTranslateNSCtoVC

Examples

This section presents a series of programming examples that illustrate basic mouse functions and important concepts involved when writing software for the mouse. There are five examples contained in this section.

- Example 11-1 explains some sample initialization procedures.
- Example 11-2 shows how information is obtained from the mouse through the ReadInputEvent procedure.
- Example 11-3 describes how a graphics cursor shape can be defined.
- Example 11-4 shows how to change a graphics cursor.
- Example 11-5 demonstrates how to turn tracking off so that the screen cursor does not track the mouse automatically.

Important Concepts

Important concepts discussed in this section include the following:

- Screen coordinate systems (Example 11-1)
- Cursor tracking (Examples 11-1 and 11-5)
- Motion rectangles (Example 11-2)

The examples contained in this section are excerpts from a C program that allows a user to sketch a drawing with the mouse. The complete sketching program follows this example section. These examples were developed using the CCGI+ Library. You could also use *Graphics.lib* to create this example; the graphics calls would be different.

Using the Mouse Buttons

Mouse buttons should be used consistently across applications. When the mouse is set up for right-handed use, the three mouse buttons are used as follows:

Left button	Mark
Middle button	Pop-up menus
Right button	Bound (or a similar function)

To be compatible with the two-button mouse, applications that make use of the middle button on the three-button mouse should also make those middle-button features accessible through the keyboard.

Screen Coordinates

Mouse procedures use two different sets of screen coordinates, as described in the subsections below.

Normalized Screen Coordinates

These coordinates are obtained with the GetIBusDevInfo procedure. For example, for one workstation type, the (x, y) coordinates are (0, 0) for the top-left corner of the screen and (32767, 22435) for the bottom-right corner of the screen.

Normalized screen coordinates are used when a high degree of precision is required. The mouse procedures themselves deal directly in normalized screen coordinates.

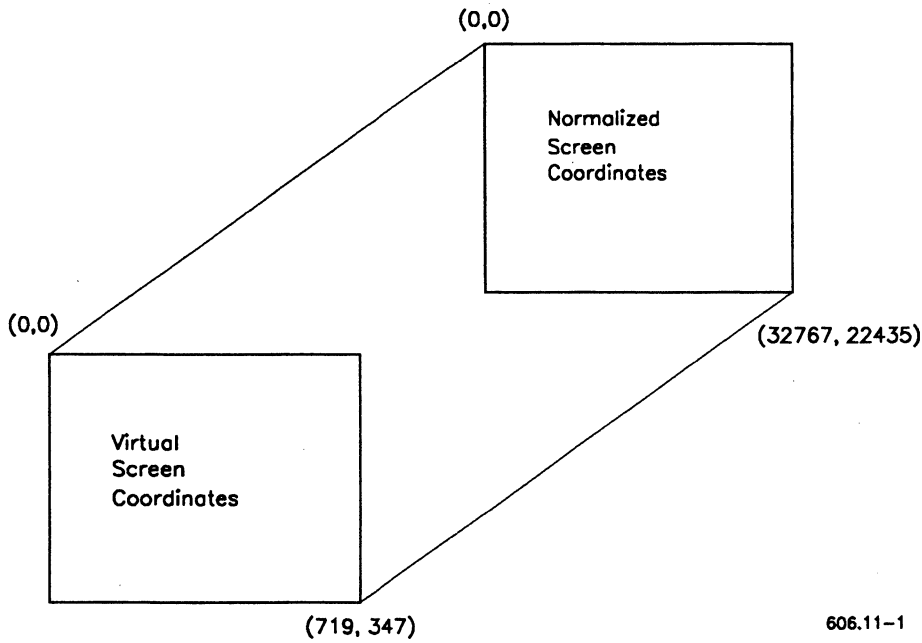
Virtual Screen Coordinates

These coordinates are application-defined coordinates. It is recommended that, for the most part, your programs use these virtual screen coordinates.

Virtual screen coordinates can, for example, be based on the number of pixels in a particular workstation's bit map. Or they can be based on any other useful division of the screen, such as (100, 80). Figure 11-1 compares normalized screen coordinates for a particular workstation with one possible set of virtual screen coordinates.

The PDTranslateNSCtoVC and PDTranslateVCtoNSC procedures can be used to convert screen coordinates from one system to the other. The object module procedures that deal in virtual screen coordinates perform this transformation for you. If you use the request-and-wait style of programming, you may want to use these procedures since the requests handle only normalized screen coordinates.

Figure 11-1. Virtual and Normalized Screen Coordinates



Setup

Example 11-1 shows a sample series of initialization procedures.

Example 11-1. Setup

```
/*
 * Set up the tracker.
 */
erc = PDSetCursorType(1);
if (erc != ercOK) return(erc);
erc = InitDrawingCursor();
if (erc != ercOK) return(erc);
erc = PDSetVirtualCoordinates (0, 0,
   ScreenAttrs.wxDeviceMax, ScreenAttrs.wyDeviceMax);
if (erc != ercOK) return(erc);
erc = PDSetTracking(TRUE);
if (erc != ercOK) return(erc);
erc = PDSetCursorPos(ScreenAttrs.wxDeviceMax / 2,
   ScreenAttrs.wyDeviceMax / 2);
.
.
.
```

`PDSetCursorType (1)`

Specifies a graphics cursor. The shape of the graphics cursor is defined in InitDrawCursor (see Example 11-3, below). On character-mapped workstations, the character cursor is a reverse video block, which cannot be changed. The graphics cursor is the default arrow or an icon defined by the programmer.

```
PDSetVirtualCoordinates (0, 0, ScreenAttrs.wxDeviceMax,
            ScreenAttrs.wyDeviceMax)
```

Sets up the virtual screen coordinates based on the raster coordinates for a particular workstation type. In this example, the (x, y) coordinates are (0, 0) for the top-left corner of the screen and (*ScreenAttrs.wxDeviceMax, ScreenAttrs.wyDeviceMax*) for the bottom-right corner of the screen (see the description of virtual screen coordinates, above, and Figure 11-1).

The raster coordinates (pixel count) for the workstation are obtained in this example by using the CGI cgi_OpenWk procedure call for the screen. This information can alternatively be obtained by using the QueryVideo service (see QueryVideo in the CTOS Procedural Interface Reference Manual). QueryVideo can be used also to obtain character coordinates for a character cursor.

```
PDSetTracking (TRUE)
```

Turns on cursor tracking. The cursor on the screen moves with the mouse. Cursor tracking is a screen attribute because only one mouse cursor can be on the screen. The cursor can, however, take on different shapes, depending on the application. When multiple windows are on the screen, the cursor thus moves across all windows. Example 11-5 shows how to turn off cursor tracking to separate movement of the screen cursor from movement of the mouse.

```
PDSetCursorPos (ScreenAttrs.wxDeviceMax / 2,
                ScreenAttrs.wyDeviceMax / 2)
```

Places the cursor in the middle of the screen. Note that this procedure uses virtual screen coordinates. Cursor tracking continues from this position.

Queries

Example 11-2 shows using the mouse to draw a line and illustrates the use of motion rectangles with ReadInputEvent.

A motion rectangle is a rectangle that is defined on the screen. When a motion rectangle has been defined and the cursor moves outside of this rectangle, an input event occurs and is returned by ReadInputEvent.

In certain applications, for example, the motion rectangle might be set to the size of a single character, 9-by-12 bits. Mouse movement within a given 9-by-12-bit area is not processed. Only when the cursor moves out of this 9-by-12-bit motion rectangle is an event returned by ReadInputEvent.

Figure 11-2 shows how movement within a motion rectangle is ignored (1). Movement *outside* of the motion rectangle, however, results in a motion rectangle event being returned by ReadInputEvent (2). PDSetMotionRectangle is called again to set another motion rectangle (3).

When a character cursor is used, the virtual screen coordinates are set according to the character map. On one type of workstation, for example, the virtual screen coordinates are (0, 0) for the top-left corner and (79, 28) for the bottom-right corner. (See QueryVideo in the *CTOS Procedural Interface Reference Manual.*) The motion rectangle would then be set to (1,1), so that the cursor movement from one character space to another would be returned, but movement within a character space would be ignored. Motion rectangles thus avoid unnecessary processing of mouse movement.

A motion rectangle event is a one-time occurrence. After the cursor moves outside of the motion rectangle, that motion rectangle is canceled. Another call to PDSetMotionRectangle must be made to set a new motion rectangle at the new position. (See Figure 11-2.)

```
PDGetCursorPos (&wOldX, &wOldY)
```

Returns the current cursor coordinates.

```
PDSetMotionRectangle (wOldX, wOldY, 1, 1)
```

Defines a motion rectangle using the current cursor coordinates as minimum values. The width and height of the motion rectangle are both 1. The motion rectangle is specified in virtual screen coordinates.

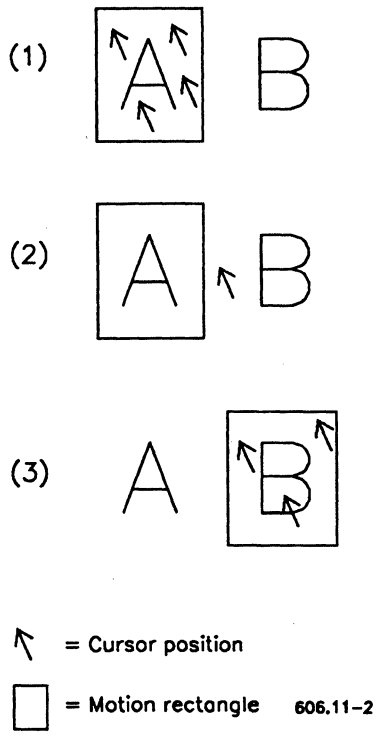
The application waits for a mouse button event, a motion rectangle event, or a keystroke. Note that after a motion rectangle event, the motion rectangle must be reset with PDSetMotionRectangle.

ReadInputEvent returns input from both the keyboard and the mouse. Note that use of ReadKbdDirect causes mouse information to be lost. If ReadKbdDirect is used, the motion rectangle may thus be lost and should be reset.

```
DrawRasterLine (wOldX, wOldY, wX, wY)
```

Draws a line from the old cursor position to the new position.

Figure 11-2. Cursor Movement Inside and Outside a Motion Rectangle



Example 11-2. Queries

```
ErcType Sketcher()

{
   Byte bChar;
   FlagType fUserHappy = TRUE, fFound;
   Word Idx;
   EventBlockType EventBlock;

   fButtonPress = FALSE;
   bButton = lNoPress;

   erc = PDGetCursorPos(&wOldX, &wOldY);
   if (erc != ercOK) return(erc);
   erc = PDSetMotionRectangle (wOldX, wOldY, 1, 1);
   if (erc != ercOK) return(erc);

   erc = cgi_LineColor(dhScreen, iColor);
   if (erc != ercOK) return(erc);
   /*
    * Loop until the user hits any key on the keyboard.
    */
   while(fUserHappy)
      {
      erc = ReadInputEvent(lWait, &EventBlock,
         lsEventBlock);
      if (erc != ercOK) return(erc);
      /*
       * Mouse button event.
       */
      if ((EventBlock.wType == lMouseButtonEvent) ||
         (EventBlock.wType == lMotionRectangleEvent))
         {
         if (EventBlock.wType == lMouseButtonEvent)
            bButton = EventBlock.bCode;
         wX = EventBlock.wX;
         wY = EventBlock.wY;
         if (EventBlock.wType == lMotionRectangleEvent)
            {
```

Example 11-2. Queries (cont.)

```
            /*
             * Reset the motion rectangle.
             */
            erc = PDSetMotionRectangle(wX, wY, 1, 1);
            if (erc != ercOK) return(erc);
            }
         fButtonPress = TRUE;
         if (bButton == lNoPress)
            fButtonPress = FALSE;
         /*
          * Is the user in the sketching area?
          */
         if ((wY < wUserMaxY) && (wY > wUserMinY) &&
            (wX > wUserMinX) && (wX < wUserMaxX))
            {
            if((bButton != lNoPress) && ((wX != wOldX) ||
               (wY != wOldY)))
               {
               erc = DrawRasterLine(wOldX,wOldY, wX,wY);
               if (erc != ercOK) return(erc);
               }
            wOldX = wX;
            wOldY = wY;
            }
         else {
            /*
             * User may be choosing a color.
             */
            .
            .
            }
         }
      else {
         /*
          * Keystroke event.
          */
         .
         .
         }
      } /* end of While (fUserHappy) */
   return(ercOK);
} /* end of Sketcher */
```

Cursor Shape

Example 11-3 defines a graphics cursor (a fountain pen).

Example 11-3. Cursor Shape

```
ErcType InitDrawingCursor ()

{
   ErcType erc = ercOK;

   DrawingCursor.wSignature = lIconSig;
   DrawingCursor.wVersion = 0;
   DrawingCursor.sIcon = 512;
   DrawingCursor.cbLine = 8;
   DrawingCursor.bWidth = 64;
   DrawingCursor.bHeight = 64;
   DrawingCursor.bxOffset = 0;
   DrawingCursor.byOffset = 0;
   DrawingCursor.Pattern[0] = 0x1;
   DrawingCursor.Pattern[4] = 0x6;
   DrawingCursor.Pattern[8] = 0x0C;
   DrawingCursor.Pattern[12] = 0x18;
   DrawingCursor.Pattern[16] = 0x38;
   DrawingCursor.Pattern[20] = 0x78;
   DrawingCursor.Pattern[24] = 0x0F8;
   DrawingCursor.Pattern[28] = 0x1E8;
   DrawingCursor.Pattern[32] = 0x3C8;
   DrawingCursor.Pattern[36] = 0x788;
   DrawingCursor.Pattern[40] = 0x1710;
   DrawingCursor.Pattern[44] = 0x3E20;
   DrawingCursor.Pattern[48] = 0x4C40;
   DrawingCursor.Pattern[52] = 0x8480;
   DrawingCursor.Pattern[56] = 0x300;
   DrawingCursor.Pattern[57] = 0x1;
   DrawingCursor.Pattern[60] = 0x180;
   DrawingCursor.Pattern[61] = 0x2;
   DrawingCursor.Pattern[64] = 0x100;
   DrawingCursor.Pattern[65] = 0x4;
   DrawingCursor.Pattern[68] = 0x200;
```

Example 11-3. Cursor Shape (cont.)

```
    DrawingCursor.Pattern[69] = 0x8;
    DrawingCursor.Pattern[72] = 0x400;
    DrawingCursor.Pattern[73] = 0x10;
    DrawingCursor.Pattern[76] = 0x800;
    DrawingCursor.Pattern[77] = 0x20;
    DrawingCursor.Pattern[80] = 0x1000;
    DrawingCursor.Pattern[81] = 0x40;
    DrawingCursor.Pattern[84] = 0x2000;
    DrawingCursor.Pattern[85] = 0x80;
    DrawingCursor.Pattern[88] = 0x4000;
    DrawingCursor.Pattern[89] = 0x100;
    DrawingCursor.Pattern[92] = 0x8000;
    DrawingCursor.Pattern[93] = 0x200;
    DrawingCursor.Pattern[97] = 0x401;
    DrawingCursor.Pattern[101] = 0x802;
    DrawingCursor.Pattern[105] = 0x1004;
    DrawingCursor.Pattern[109] = 0x2008;
    DrawingCursor.Pattern[113] = 0x4010;
    DrawingCursor.Pattern[117] = 0x8060;
    DrawingCursor.Pattern[121] = 0x80;
    DrawingCursor.Pattern[122] = 0x1;
    DrawingCursor.Pattern[125] = 0x100;
    DrawingCursor.Pattern[126] = 0x2;
    DrawingCursor.Pattern[129] = 0x600;
    DrawingCursor.Pattern[130] = 0x4;
    DrawingCursor.Pattern[133] = 0x800;
    DrawingCursor.Pattern[134] = 0x8;
    DrawingCursor.Pattern[137] = 0x1000;
    DrawingCursor.Pattern[138] = 0x10;
    DrawingCursor.Pattern[141] = 0x6000;
    DrawingCursor.Pattern[142] = 0x10;
    DrawingCursor.Pattern[145] = 0x8000;
    DrawingCursor.Pattern[146] = 0x11;
    DrawingCursor.Pattern[150] = 0x1E;

    CheckErc (PDLoadCursor (&DrawingCursor, 544, 3));

} /* end of InitDrawingCursor */
```

Changing the Graphics Cursor

Example 11-4 shows how to change the graphics cursor.

Example 11-4. Changing the Graphics Cursor

```
/* Open cursor icon file. */
.
.
.
CheckErc (PDSetCursorDisplay (FALSE));
CheckErc (PDReadIconFile (fh, &Cursor, sCursor,
          &WorkArea, sWorkArea, &cbRet));
CheckErc (PDLoadCursor (&Cursor, sCursor, bType));
CheckErc (PDSetCursorDisplay (TRUE));
```

The procedures used in this code are defined below:

`PDSetCursorDisplay (FALSE)`

Turns off the cursor display.

`PDReadIconFile (fh, &Cursor, sCursor, &WorkArea, sWorkArea, &cbRet)`

Reads the file for the new icon.

`PDLoadCursor (&Cursor, sCursor, bType)`

Changes the cursor to the new icon.

`PDSetCursorDisplay (TRUE)`

Turns on the new cursor display.

Note: *This example assumes that Cursor is the same memory area for the old and new cursor shapes. Therefore it is necessary to turn off the cursor display before loading the new cursor. If the old and new cursor shapes are in different memory areas, you need not turn off the cursor display before loading the new cursor.*

Cursor Movement

This example shows how an application can separate mouse tracking from cursor movement. Here, the programmer wants to have the cursor move in discrete jumps as if bound to a grid. Initialization is similar to Example 11-1, except that instead of calling PDSetTracking to turn on the cursor display and start tracking movement of the mouse with the cursor, PDSetCursorDisplay is called to turn on the cursor only. The second part of this example is similar to Example 11-2. Underlined portions of the program show the changes necessary to Examples 11-1 and 11-2.

Example 11-5. Cursor Movement

```
erc = PDSetCursorType(1);
if (erc != ercOK) return(erc);
erc = PDSetVirtualCoordinates (0, 0,
   ScreenAttrs.wxDeviceMax, ScreenAttrs.wyDeviceMax);
if (erc != ercOK) return(erc);
erc = (PDSetCursorDisplay (TRUE);
erc = PDSetCursorPos(ScreenAttrs.wxDeviceMax / 2,
   ScreenAttrs.wyDeviceMax / 2);

.
.
.
/*
 * Loop until the user hits any key on the keyboard.
 */
while(fUserHappy)
   {
   erc = ReadInputEvent(lWait, &EventBlock,
      lsEventBlock);
   if (erc != ercOK) return(erc);

   /*
    * Mouse button event.
    */
```

Example 11-5. Cursor Movement (cont.)

```
if ((EventBlock.wType == lMouseButtonEvent) ||
   (EventBlock.wType == lMotionRectangleEvent))
   {
   if (EventBlock.wType == lMouseButtonEvent)
      bButton = EventBlock.bCode;
   wX = EventBlock.wX;
   wY = EventBlock.wY;
   /*
    * Calculate the on-grid values for x and y
    */
   wX = ((wX + (wGridValue / 2)) / wGridValue) *
        wGridValue;
   wY = ((wY + (wGridValue / 2)) / wGridValue) *
        wGridValue;

   /*
    * Check to see if we need to move the cursor
    */
   if ((wX != wOldX || (wY != wOldY))
      CheckErc(PDSetCursorPos(wX, wY));

   if (EventBlock.wType == lMotionRectangleEvent)
      {
      /* Reset the motion rectangle. */

      erc = PDSetMotionRectangle(wX, wY, 1, 1);
      if (erc != ercOK) return(erc);
```

Troubleshooting

If problems occur with loading your own cursor, you may have forgotten to reverse the order of the bits of the cursor image when loading them into screen memory. See the PDLoadCursor procedure in *CTOS Procedural Interface Reference Manual.*

If it appears that you are not receiving motion rectangle events, check to be sure that the procedures you are calling are preserving motion rectangles. If they are canceling motion rectangles, reissue SetMotionRectangle.

The examples in this section use the CCGI+ Library. If you are using an older version of the Graphics Library, you may notice fragments of the cursor on the screen or fragments of graphics. If this occurs, correct the problem by preceding the calls to the Graphics Library with PDSetCursorDisplay(false), and ending the series of Graphics Library calls with PDSetCursorDisplay(true), as illustrated below:

```
PDSetCursorDisplay (false);
.
./* Calls to Graphics Library */
.
PDSetCursorDisplay (true);
```

Sample Mouse Program

Example 11-6 is a sample program that allows a user to sketch using a pointing device.

Example 11-6. Sketcher.c

```
#include <stdio.h>
#include <stdlib.h>

#define CheckErc
#define ResetFrame
#include <CTOS.Lib.h>

#include "cgi_User.h"     /* Comes with CGI.lib */

DWord dwGraphicsMemory = 32768L;
unsigned int wHPixels;
unsigned int wVPixels;

cgi_InitParamType InitParam;
cgi_OpenWkRetType ScreenAttrs;
cgi_OpenWkRetType OffScreenAttrs;
cgi_OpenWkParamType OffScreenParam;
cgi_OpenWkParamType DumpFileParam;
cgi_OpenWkRetType DumpFileAttrs;
Word dhScreen;
```

Example 11-6. Sketcher.c (cont.)

```
#define MaxColor 9          /* maximum colors used */
rgRGBColorType ColorPalette[MaxColor];

/*
 *Type declarations
 */
typedef struct {
   Word wSignature;
   Word wVersion;
   Word sIcon;
   Word cbLine;
   Byte bWidth;
   Byte bHeight;
   Byte bxOffset;
   Byte byOffset;

   Byte bFlag;
   Byte rgbReserved[3];
   Byte sbName[16];
   Word Pattern[256];
} IconHeaderType;

typedef struct {
   Word wType;
   Byte bCode;
   Word wX;
   Word wY;
} EventBlockType;

typedef struct {
   Word X1;
   Word Y1;
   Word X2;
   Word Y2;
} rgPaletteType;
```

Example 11-6. Sketcher.c (cont.)

```
#define lbFinish                4
#define lbClearFKey             0x1F    /* F10 key */
#define lWhite                  0
#define lBlack                  1
#define lYellow                 2
#define lGreen                  3
#define lCyan                   4
#define lBlue                   5
#define lMagenta                6
#define lRed                    7
#define lKeyboardEvent          0x20
#define lMotionRectangleEvent   0x60
#define lMouseButtonEvent       0x80
#define lIconSig                0x4349
#define lsEventBlock            7
#define lWait                   0
#define lNoPress                0
#define lBoxStartOffset         20
#define lBoxEndOffset           50
#define lBoxHeight              10

Word wOldX, wOldY, wX, wY, wUserMaxX, wUserMaxY,
   wUserMinX, wUserMinY, wCurY, iColor;
ErcType erc;
Byte bButton;
FlagType fButtonPress, fBitmap;
IconHeaderType DrawingCursor;
rgPaletteType rgPalette[8];
```

Example 11-6. Sketcher.c (cont.)

```
/*  Mouse calls */

extern ErcType _plm PDGetCursorPos(Pointer pwXPosRet,
  Pointer pwYPosRet);
extern ErcType _plm PDLoadCursor(IconHeaderType
            *pCursorShape, Word sCursorShape, Byte bType);
extern ErcType _plm PDSetCursorPos(Word wXPos, Word wYPos);
extern ErcType _plm PDSetCursorType(Byte bType);
extern ErcType _plm PDSetMotionRectangle(Word wXMin,
   Word wYMin, Word wDX, Word wDY);
extern ErcType _plm PDSetTracking(FlagType fOn);
extern ErcType _plm PDSetVirtualCoordinates(Word wXMin,
   Word wYMin, Word wXMax, Word wYMax);
extern ErcType _plm ReadInputEvent(Word wMode,
   EventBlockType *pEventBlock, Word sEventBlock);
extern ErcType _plm cgi_FillColor(Word dh, Word Idx);
extern ErcType _plm cgi_EdgeColor(Word dh, Word Idx);
extern ErcType _plm CGI_Initialize (DWord dwGraphicsMemory,
   cgi_InitParamType *InitParam, Word sInitParam);
extern ErcType _plm cgi_InteriorStyle (Word dh, Word
            wStyle);
extern ErcType _plm cgi_LineColor(Word dh, Word Color);
extern ErcType _plm CGI_OpenWk (Pointer Params, Word
                  sParams,
   Word *dhScreen, cgi_OpenWkRetType *ScreenAttrs,
   Word sScreenAttrs);
extern ErcType _plm cgi_Polyline(Word dh, Word Points,
   Word *rgwPoints);
extern ErcType _plm cgi_PrepareViewSurface(Word dh,
   Word HardCopy);
extern ErcType _plm CGI_VDCExtent (Word dh, Word X1,
   Word Y1,  Word X2, Word Y2);
```

Example 11-6. Sketcher.c (cont.)

```
/*
 * SetColorPalette
 *
 * This procedure sets up the colors for sketching
 */
ErcType     SetColorPalette()
{
   ErcType erc = ercOK;

   ColorPalette[lWhite].wRed = 1000; /* background */
   ColorPalette[lWhite].wGreen = 1000;
   ColorPalette[lWhite].wBlue =  1000;

   ColorPalette[lBlack].wRed = 0;
   ColorPalette[lBlack].wGreen = 0;
   ColorPalette[lBlack].wBlue = 0;

   ColorPalette[lGreen].wRed = 0;
   ColorPalette[lGreen].wGreen =1000;
   ColorPalette[lGreen].wBlue =  0;

   ColorPalette[lBlue].wRed = 0;
   ColorPalette[lBlue].wGreen = 0;
   ColorPalette[lBlue].wBlue = 1000;

   ColorPalette[lYellow].wRed = 1000;
   ColorPalette[lYellow].wGreen =1000;
   ColorPalette[lYellow].wBlue =  0;

   ColorPalette[lCyan].wRed = 0;
   ColorPalette[lCyan].wGreen = 1000;
   ColorPalette[lCyan].wBlue =1000;

   ColorPalette[lMagenta].wRed = 1000;
   ColorPalette[lMagenta].wGreen = 0;
   ColorPalette[lMagenta].wBlue = 1000;

   ColorPalette[lRed].wRed = 1000;
   ColorPalette[lRed].wGreen = 0;
   ColorPalette[lRed].wBlue = 0;
```

Example 11-6. Sketcher.c (cont.)

```
    /* Set the alpha color same as foreground graphics
     * color.
     */
    ColorPalette[8].wRed =   ColorPalette[1].wRed;
    ColorPalette[8].wGreen = ColorPalette[1].wGreen;
    ColorPalette[8].wBlue =  ColorPalette[1].wBlue;

    erc = cgi_ColorTable(dhScreen, 0, MaxColor,
          ColorPalette);
    return erc;
}
/*
 * FillRasterRectangle
 *
 * This procedure fills a rectangle specified in
 * raster coordinates.
 */
ErcType FillRasterRectangle(Word wX1, Word wY1,
    Word wX2, Word wY2, Byte bFillType)
{
    ErcType erc;
    #define lsPoints 4
    Word rgwPoints[8];

    rgwPoints[0] = wX1;
    rgwPoints[1] = wY1;
    rgwPoints[2] = wX2;
    rgwPoints[3] = wY1;
    rgwPoints[4] = wX2;
    rgwPoints[5] = wY2;
    rgwPoints[6] = wX1;
    rgwPoints[7] = wY2;

    erc = cgi_Polygon(dhScreen, lsPoints, rgwPoints);

    return(erc);
}
```

Example 11-6. Sketcher.c (cont.)

```
/*
 * DrawRasterLine
 *
 * This procedure draws a line specified in raster
 * coordinates.
 */
ErcType DrawRasterLine(Word wX1, Word wY1, Word wX2,
   Word wY2)

{
   ErcType erc;
   #define lsLinePoints 2
   Word rgwPoints[4];

   rgwPoints[0] = wX1;
   rgwPoints[1] = wY1;
   rgwPoints[2] = wX2;
   rgwPoints[3] = wY2;

   erc = cgi_Polyline(dhScreen, lsLinePoints, rgwPoints);
   return(erc);
}

/*
 * InitDrawingCursor
 *
 * This procedure initializes the sketch cursor.
 */
ErcType InitDrawingCursor()

{
   ErcType erc = ercOK;

   DrawingCursor.wSignature = lIconSig;
   DrawingCursor.wVersion = 0;
   DrawingCursor.sIcon = 512;
   DrawingCursor.cbLine = 8;
   DrawingCursor.bWidth = 64;
   DrawingCursor.bHeight = 64;
   DrawingCursor.bxOffset = 0;
   DrawingCursor.byOffset = 0;
   DrawingCursor.Pattern[0] = 0x1;
```

Example 11-6. Sketcher.c (cont.)

```
DrawingCursor.Pattern[4] = 0x6;
DrawingCursor.Pattern[8] = 0x0C;
DrawingCursor.Pattern[12] = 0x18;
DrawingCursor.Pattern[16] = 0x38;
DrawingCursor.Pattern[20] = 0x78;
DrawingCursor.Pattern[24] = 0x0F8;
DrawingCursor.Pattern[28] = 0x1E8;
DrawingCursor.Pattern[32] = 0x3C8;
DrawingCursor.Pattern[36] = 0x788;
DrawingCursor.Pattern[40] = 0x1710;
DrawingCursor.Pattern[44] = 0x3E20;
DrawingCursor.Pattern[48] = 0x4C40;
DrawingCursor.Pattern[52] = 0x8480;
DrawingCursor.Pattern[56] = 0x300;
DrawingCursor.Pattern[57] = 0x1;
DrawingCursor.Pattern[60] = 0x180;
DrawingCursor.Pattern[61] = 0x2;
DrawingCursor.Pattern[64] = 0x100;
DrawingCursor.Pattern[65] = 0x4;
DrawingCursor.Pattern[68] = 0x200;
DrawingCursor.Pattern[69] = 0x8;
DrawingCursor.Pattern[72] = 0x400;
DrawingCursor.Pattern[73] = 0x10;
DrawingCursor.Pattern[76] = 0x800;
DrawingCursor.Pattern[77] = 0x20;
DrawingCursor.Pattern[80] = 0x1000;
DrawingCursor.Pattern[81] = 0x40;
DrawingCursor.Pattern[84] = 0x2000;
DrawingCursor.Pattern[85] = 0x80;
DrawingCursor.Pattern[88] = 0x4000;
DrawingCursor.Pattern[89] = 0x100;
DrawingCursor.Pattern[92] = 0x8000;
DrawingCursor.Pattern[93] = 0x200;
DrawingCursor.Pattern[97] = 0x401;
DrawingCursor.Pattern[101] = 0x802;
DrawingCursor.Pattern[105] = 0x1004;
DrawingCursor.Pattern[109] = 0x2008;
DrawingCursor.Pattern[113] = 0x4010;
DrawingCursor.Pattern[117] = 0x8060;
DrawingCursor.Pattern[121] = 0x80;
DrawingCursor.Pattern[122] = 0x1;
DrawingCursor.Pattern[125] = 0x100;
```

Example 11-6. Sketcher.c (cont.)

```
    DrawingCursor.Pattern[126] = 0x2;
    DrawingCursor.Pattern[129] = 0x600;
    DrawingCursor.Pattern[130] = 0x4;
    DrawingCursor.Pattern[133] = 0x800;
    DrawingCursor.Pattern[134] = 0x8;
    DrawingCursor.Pattern[137] = 0x1000;
    DrawingCursor.Pattern[138] = 0x10;
    DrawingCursor.Pattern[141] = 0x6000;
    DrawingCursor.Pattern[142] = 0x10;
    DrawingCursor.Pattern[145] = 0x8000;
    DrawingCursor.Pattern[146] = 0x11;
    DrawingCursor.Pattern[150] = 0x1E;

    erc = PDLoadCursor(&DrawingCursor, 544, 3);
    return(erc);
}

/*
 * Sketcher
 *
 * This procedure handles the mouse events and lets the
 * user draw in color.
 */
ErcType Sketcher()

{
    Byte bChar;
    FlagType fUserHappy = TRUE, fFound;
    Word Idx;
    EventBlockType EventBlock;

    fButtonPress = FALSE;
    bButton = lNoPress;

    erc = PDGetCursorPos(&wOldX, &wOldY);
    if (erc != ercOK) return(erc);
    erc = PDSetMotionRectangle (wOldX, wOldY, 1, 1);
    if (erc != ercOK) return(erc);

    erc = cgi_LineColor(dhScreen, iColor);
    if (erc != ercOK) return(erc);
```

Example 11-6. Sketcher.c (cont.)

```
/*
 * Loop until the user hits any key on the keyboard.
 */
while(fUserHappy)
   {
   erc = ReadInputEvent(lWait, &EventBlock,
      lsEventBlock);
   if (erc != ercOK) return(erc);

   /*
    * Mouse button event.
    */
   if ((EventBlock.wType == lMouseButtonEvent) ||
      (EventBlock.wType == lMotionRectangleEvent))
      {
      if (EventBlock.wType == lMouseButtonEvent)
         bButton = EventBlock.bCode;
      wX = EventBlock.wX;
      wY = EventBlock.wY;

      if (EventBlock.wType == lMotionRectangleEvent)
         {
         /*
          * Reset the motion rectangle.
          */
         erc = PDSetMotionRectangle(wX, wY, 1, 1);
         if (erc != ercOK) return(erc);
         }
      fButtonPress = TRUE;
      if (bButton == lNoPress)
         fButtonPress = FALSE;

      /*
       * Is the user in the sketching area?
       */
      if ((wY < wUserMaxY) && (wY > wUserMinY) &&
         (wX > wUserMinX) && (wX < wUserMaxX))
         {
         if((bButton != lNoPress) && ((wX != wOldX) ||
            (wY != wOldY)))
            {
            erc = DrawRasterLine(wOldX,wOldY, wX,wY);
```

Example 11-6. Sketcher.c (cont.)

```
            if (erc != ercOK) return(erc);
            }
         wOldX = wX;
         wOldY = wY;
         }
      else {
         /*
          * User may be choosing a color.
          */
         if ((wX >= wUserMaxX) &&
            (bButton !=lNoPress))
            {
            /*
             * Check to see if user choosing a color.
             */
            fFound = FALSE;
            Idx = 0;
            while ((!fFound) && (Idx < 8))
               {
               if ((wX > rgPalette[Idx].X1) &&
                  (wX < rgPalette[Idx].X2) &&
                  (wY > rgPalette[Idx].Y1) &&
                  (wY < rgPalette[Idx].Y2))
                  fFound = TRUE;
               else Idx++;
               }

            if (fFound && ((Idx) != iColor))
               {
               /*
                * Remove current color indicator.
                */
               erc = cgi_FillColor(dhScreen, 0);
               if (erc != ercOK) return(erc);
               erc = cgi_EdgeColor(dhScreen, 0);
               if (erc != ercOK) return(erc);
               erc = FillRasterRectangle(
                  rgPalette[iColor].X1 - 6,
                  rgPalette[iColor].Y1 + 3,
                  rgPalette[iColor].X1 - 3,
                  rgPalette[iColor].Y1 + 6,0);
```

Example 11-6. Sketcher.c (cont.)

```
                if (erc != ercOK) return(erc);
                iColor = Idx;
                if (erc != ercOK) return(erc);
                iColor = Idx;                    /*
                 * Show new color.
                 */
                erc = cgi_FillColor(dhScreen, iColor);
                if (erc != ercOK) return(erc);
                erc = cgi_EdgeColor(dhScreen, lBlack);
                if (erc != ercOK) return(erc);
                erc = FillRasterRectangle(
                   rgPalette[Idx].X1 - 6,
                   rgPalette[Idx].Y1 + 3,
                   rgPalette[Idx].X1 - 3,
                   rgPalette[Idx].Y1 + 6, 0);
                if (erc != ercOK) return(erc);
                erc = cgi_LineColor(dhScreen, iColor);
                if (erc != ercOK) return(erc);
                }
             }
          }
       }
    else {
       /*
        * Keystroke event.
        */
       if (EventBlock.wType == lKeyboardEvent)
          {
          bChar = EventBlock.bCode;
          if (bChar == lbClearFKey)
             {
             /* Clear the sketching area. */
             erc = cgi_EdgeColor(dhScreen, lWhite);
             if (erc != ercOK) return(erc);
             erc = cgi_FillColor(dhScreen, lWhite);
             if (erc != ercOK) return(erc);

             erc = FillRasterRectangle(
                   wUserMinX+1, wUserMinY+1,
                   wUserMaxX-1, wUserMaxY-1, 0);
             if (erc != ercOK) return(erc);
             }
```

Example 11-6. Sketcher.c (cont.)

```
            else {
               if (bChar == lbFinish) fUserHappy = FALSE;
               }
            }
         }
      } /* end of While (fUserHappy) */
   return(ercOK);
}

/*
 * MainProg
 *
 * This is the main procedure for the sketcher and
 * handles initialization.
 */
ErcType MainProg()

{
   Word Idx;

   erc = ResetFrame(0);
   erc = ResetFrame(1);
   erc = ResetFrame(2);

   iColor = 1;
   erc = CGI_Initialize (dwGraphicsMemory, &InitParam,
         sizeof(InitParam));
   if (erc != ercOK) return (erc);

   /* Open the screen device. */
   erc = CGI_OpenWk (NULL, 0, &dhScreen, &ScreenAttrs,
         sizeof(ScreenAttrs));
   if (erc != ercOK) return(erc);
   /*
    * Set the extent to match the physical resolution of
    * the display.
    */
   erc = CGI_VDCExtent (dhScreen,
         0, ScreenAttrs.wyDeviceMax,
         ScreenAttrs.wxDeviceMax, 0);
   if (erc != ercOK) return (erc);
```

Example 11-6. Sketcher.c (cont.)

```
erc = cgi_PrepareViewSurface(dhScreen, 0);
if (erc != ercOK) return(erc);
erc = cgi_PrepareViewSurface(dhScreen, 0);
if (erc != ercOK) return(erc);    erc = SetColorPalette();
if (erc != ercOK) return(erc);

/*
 * Make a box around the usable sketching area leaving
 * room around the edges.
 */
wUserMaxX = ScreenAttrs.wxDeviceMax -
           (2*lBoxStartOffset + lBoxEndOffset);
wUserMaxY = ScreenAttrs.wyDeviceMax - lBoxStartOffset;
wUserMinX = lBoxStartOffset/2;
wUserMinY = wUserMinX;

iColor = lBlack;
erc = cgi_FillColor(dhScreen, 0);
if (erc != ercOK) return(erc);
erc = FillRasterRectangle(wUserMinX, wUserMinY,
      wUserMaxX, wUserMaxY, 0);
if (erc != ercOK) return(erc);

wCurY = wUserMinY + lBoxStartOffset;
erc = cgi_InteriorStyle(dhScreen, lSolid);
if (erc != ercOK) return(erc);
/*
 * Draw color boxes
 */
for (Idx = 0; Idx < 8; Idx++) {
   rgPalette[Idx].X1 = wUserMaxX + lBoxStartOffset;
   rgPalette[Idx].X2 = wUserMaxX + lBoxEndOffset;
   rgPalette[Idx].Y1 = wCurY + lBoxHeight;
   rgPalette[Idx].Y2 = rgPalette[Idx].Y1 + lBoxHeight;
   wCurY = rgPalette[Idx].Y2;

   erc = cgi_FillColor(dhScreen, Idx);
  if (erc != ercOK) return(erc);
   erc = cgi_EdgeColor(dhScreen, lBlack);
   if (erc != ercOK) return(erc);
```

Example 11-6. Sketcher.c (cont.)

```
      erc = FillRasterRectangle(rgPalette[Idx].X1,
         rgPalette[Idx].Y1, rgPalette[Idx].X2,
         rgPalette[Idx].Y2, 0);
      if (erc != ercOK) return(erc);
      }
   /*
    * Set up 'dot' for active color
    */
   erc = cgi_FillColor(dhScreen, iColor);
   if (erc != ercOK) return(erc);
   erc = cgi_EdgeColor(dhScreen, lBlack);
   erc = FillRasterRectangle(rgPalette[iColor].X1-6,
      rgPalette[iColor].Y1+3, rgPalette[iColor].X1-3,
      rgPalette[iColor].Y1 + 6, 0);
   if (erc != ercOK) return(erc);

   /*
    * Set up the tracker.
    */
   erc = PDSetCursorType(1);
   if (erc != ercOK) return(erc);
   erc = InitDrawingCursor();
   if (erc != ercOK) return(erc);
   erc = PDSetVirtualCoordinates (0, 0,
      ScreenAttrs.wxDeviceMax, ScreenAttrs.wyDeviceMax);
   if (erc != ercOK) return(erc);
   erc = PDSetTracking(TRUE);
   if (erc != ercOK) return(erc);
   erc = PDSetCursorPos(ScreenAttrs.wxDeviceMax / 2,
      ScreenAttrs.wyDeviceMax / 2);
   return(erc);
}
/*
 * MainLine
 */
void main()

{
   CheckErc(MainProg());
   CheckErc(Sketcher());
}
```

Section 12
Queue Manager

Introduction to Queue Manager

The queue management facility maintains disk-based queue entry files. Throughout the rest of this section, queue entry files are called queues. Queues are used to communicate information among programs within a workstation, between workstations, or across the network. Because queues are disk based, their contents are preserved across system reboot or power failure. Note that under similar circumstances, when interprocess communication (IPC) or inter-CPU communication (ICC) is used, such data preservation does not occur.

Each queue contains information for a single type of processing, such as spooled printing, 2780/3780 BSC remote job entry (RJE), or Systems Network Architecture (SNA) RJE. This information is created, accessed, and modified by programs that call queue management operations.

To use queues, you must install the Queue Manager command. The Queue Manager can be installed only on a server or a standalone workstation. Queues can be defined by the system administrator before the Queue Manager is installed or at any time after installation by programs that call the AddQueue operation. Each queue must be assigned a unique name and file specification.

What Is the Queue Manager?

The Queue Manager is a system service that maintains queues. In this capacity, it acts as a facilitator of queue activities generated by programs that make calls to queue management operations.

Run Files

The Queue Manager consists of two run files:

- *InstallQMgr.run* installs and deinstalls the Queue Manager.
- *QueueMgr.run* is the Queue Manager system service.

Installation and Deinstallation

The Queue Manager can be installed only on a server or on a standalone workstation.

In a cluster configuration, the Queue Manager must be installed at the server. Programs that use the queue management facility, however, can be installed at client workstations as well as at the server. In addition, multiple programs on different client workstations can access the same queue simultaneously.

To install the Queue Manager, you can use

- The system initializationn JCL file when the system is bootstrapped. (See the *CTOS System Administration* for details.)
- The Executive **Install Queue Manager** command. This command allows you to configure use of the Queue Manager for greater flexibility. (See the *CTOS Executive Reference Manual* and the *CTOS System Administration Guide* for details.)
- The Print Manager. (See the *CTOS Administration Guide* for details.)

After the Queue Manager is installed, programs can call queue management operations to perform such functions as adding or deleting queue entries, setting queue entries to be in service, or returning queue status information. The Queue Manager acts as a facilitator in accepting parameters passed to it to carry out these functions.

The Queue Manager can be deinstalled only at the server or the standalone workstation; it cannot be deinstalled from a client workstation. To deinstall the Queue Manager, you can use either the Executive command, **Deinstall Queue Manager**, or your program can call the DeinstallQueueManager operation. (For details on the **Deinstall Queue Manager** command, see the *CTOS Executive Reference Manual.*)

Functional Groups of Queue Management Operations

There are three functional groups of queue management operations that can be used by programs: queue manipulation operations, queue server operations, and client operations. A program using queue manipulation operations creates, maintains, and removes queues. A program using queue server operations processes queue entries. (Note that the term *queue server*, which refers to the group of queue management operations that control the queues, is different from the term server, which refers to the workstation server for a cluster.) A client program uses the client operations to submit entries to the queue for processing.

The queue management operations discussed below are listed and summarized in the next subsection.

Queue Manipulation Operations

The queue management operations provide a means of managing queues (not queue entries) and obtaining information about them.

These operations are:

- AddQueue
- CleanQueue
- RemoveQueue
- GetQmStatus

The AddQueue operation defines a queue dynamically and returns a queue handle. (See "Defining Queues Dynamically," later in this section.) The queue handle can be used in a subsequent call to the CleanQueue operation to reset the queue to its initial (empty) state or a call to RemoveQueue to delete the queue from the Queue Manager's list of recognized queues; the queue file, however, remains on disk.

Information about all queues of a given type can be obtained by calling GetQmStatus and specifying the queue type.

Queue Server Operations

A program can submit requests to the Queue Manager to process entries, such as when printing or transmitting files, by using the queue server operations. These operations enable a program to perform the following functions:

- Establish itself as the queue(s) server
- Mark queue entries to obtain exclusive access to those entries
- Unmark queue entries to relinquish exclusive access to those entries
- Process marked entries in the specified queue(s)
- Remove or reschedule marked queue entries
- Rewrite queue entries

Note: *Although some of the more commonly known queue servers also are system services, for example, the spooler and SNA entry (RJE), application programs as well can serve queues. The only requirement is that the program call EstablishQueueServer.*

Establishing Queue Servers

A program must establish itself as a queue server for the specified queue(s) with the EstablishQueueServer operation before it can use any of the queue server operations for marking or unmarking queues.

EstablishQueueServer enables the Queue Manager to keep a count of the number of programs serving each queue. The Queue Manager checks the count of queue servers before adding entries to a queue. If no queue servers are active, a client may elect not to queue a new entry.

Marking Queue Entries

The queue server obtains exclusive access to a queue entry by marking the entry as being in use. Marking is accomplished by using either the MarkNextQueuedEntry operation or the MarkKeyedQueueEntry operation. MarkNextQueuedEntry specifies the next available queue entry, whereas MarkKeyedQueueEntry specifies a particular entry.

The marking operations prevent interference among multiple queue servers serving a single queue. While a queue entry is marked, it is not returned in subsequent marking operations.

Unmarking Queue Entries

A queue sever program can unmark a queue entry (making it available for processing by other queue servers) by calling the UnmarkQueueEntry operation. Queue entries are also unmarked when:

- The Queue Manager is installed or deinstalled.
- A queue server terminates operation for any reason, including malfunction of a client workstation. When a queue server terminates, the Queue Manager searches all queues affected and unmarks any queue entries marked by servers from the malfunctioning workstation.
- A queue server elects to discontinue serving a queue and issues a TerminateQueueServer operation. The Queue Manager decrements the count of active queue servers for that queue and resets all entries previously marked by the terminating server. It also returns any blocked MarkNextQueueEntry requests.

Processing Queue Entries

Once a queue server has marked a queue entry as in use, it can then process the entry. The information included in the queue entry is used to determine what is to be done. Queue entries are discussed later in this section.

Removing and Rescheduling Queue Entries

The queue server can remove a queue entry from the specified queue with the RemoveMarkedQueueEntry and RescheduleMarkedQueueEntry operations. RescheduleMarkedQueueEntry also can be used to reschedule entries in a specified queue.

Rewriting Queue Entries

The queue server can rewrite a queue entry with a new entry in a specified queue with the RewriteMarkedQueueEntry operation. Using this operation allows the queue server to do such things as update a field in a queue entry.

Client Operations

A program can submit requests to the Queue Manager for processing services, such as for printing or file transmission, by using client operations. These operations enable a program to perform the following queue entry functions:

- Add an entry to a queue
- Delete previously queued entries
- Obtain a list of entries queued

Note: *Any program requesting queue entry processing can call the client operations. Such a program is commonly called a client even though the queue server, as noted earlier, is not necessarily a system service.*

Adding an Entry to a Queue

A program adds an entry to the specified queue with the AddQueueEntry operation. To do so, the program specifies information, including

- A queue name that must correspond to an already created queue.
- The memory address of a buffer containing the queue entry.
- A priority level (0 through 9, with 0 the highest) at which the entry is queued.
- An optional date/time specification for the earliest time the entry is made available for service. (For details, see "Queue Entry Processing Order," later in this section.)
- An optional repeat time interval for making the queue entry available again after it has been used. The repeat time interval is added to the time specification for servicing the entry. (For details, see "Queue Entry Processing Order," later in this section.)

When requested to add a new entry to the queue, the Queue Manager checks the number of active queue servers . If no queue servers are actively serving the queue, the Queue Manager sets a flag accordingly. In this case, some clients may elect not to queue a new entry.

Deleting an Entry

A client removes a specific queue entry from the queue with the RemoveKeyedQueueEntry operation. The queue entry is identified by one or two key fields.

A key is a particular field or combination of fields in a data record upon which the search process is performed. The RemoveKeyedQueueEntry operation can specify that up to two key fields must match corresponding fields in the queue entry before the queue entry is removed. (For details, see "Referencing Queues and Queue Entries," later in this section.)

Reading Queue Entries

A client reads queue entries with the ReadKeyedQueueEntry or the ReadNextQueueEntry operation. ReadNextQueueEntry typically is used by a program not having exclusive access to the queue to list the contents of all queue entries for processing purposes.

The client specifies the queue name, queue entry handle, and memory addresses of buffers to which the queue entry and Queue Status Block are returned. (For details on the queue entry handle and the Queue Status Block, see "Referencing Queues and Queue Entries," later in this section.)

Summary of Queue Manager Operations

The queue management operations are described below. (See the *CTOS Procedural Interface Reference Manual* for a complete description of each operation.)

AddQueue

Activates a new queue.

AddQueueEntry

Adds an entry to the specified queue for processing by the appropriate queue server.

CleanQueue

Resets a queue to empty.

DeInstallQueueManager

Terminates operation of the Queue Manager and frees its memory partition.

EstablishQueueServer

Establishes a program as a queue server for the specified queue.

GetQmStatus

Interrogates the Queue Manager about usage statistics, as well as the queues of the specified type.

MarkKeyedQueueEntry

Locates the first unmarked entry in the specified queue with up to two key fields equal to the values specified, marks it as being in use, reads it into a buffer, and returns a queue entry handle for use in a subsequent RemoveMarkedQueueEntry operation.

MarkNextQueueEntry

Reads the first unmarked entry in the specified queue into a buffer, marks it as being in use, and returns a queue entry handle. Entries are marked in order of priority.

ReadKeyedQueueEntry

Obtains the first queue entry in the specified queue with up to two key fields equal to the values specified, reads it into a buffer, and returns the Queue Status Block.

ReadNextQueueEntry

Reads an entry from the specified queue into a buffer and returns the queue entry handle of the next queue entry.

RemoveKeyedQueueEntry

Locates an unmarked entry in the specified queue with up to two key fields equal to the values specified and removes it from the queue.

RemoveMarkedQueueEntry

Removes a previously marked entry from the specified queue.

RemoveQueue

Removes a queue dynamically.

RescheduleMarkedQueueEntry

Removes a previously marked entry from the specified queue or reschedules the entry if it is associated with a repeat time interval.

RewriteMarkedQueueEntry

Rewrites the specified marked queue entry with a new queue entry.

TerminateQueueServer

Notifies the Queue Manager that a queue server is no longer serving the specified queue.

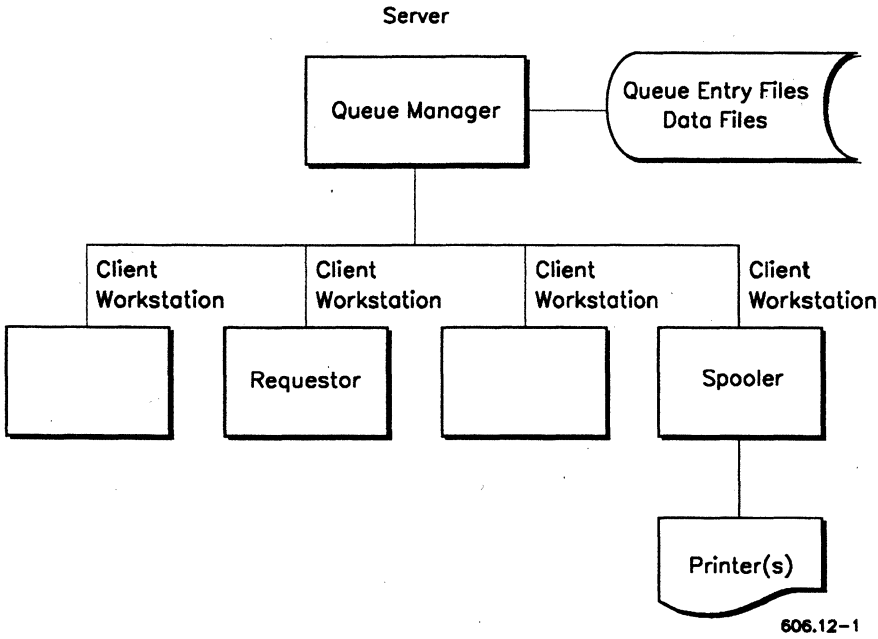
UnmarkQueueEntry

Resets the specified queue entry as unmarked (not in use).

Queue Manager Configuration

Figure 12-1 shows an example of a cluster configuration with the queue management facility, a client, and a queue server (in this case, the spooler, which is also a system service).

Figure 12-1. Configuration with the Queue Management Facility



Queues

A queue contains information for a single type of processing such as spooled printing or RJE.

Format

A queue consists of a 512-byte header record followed by a series of queue entry records.

- The queue file header contains all the data that the Queue Manager needs to control the file.
- Each queue entry consists of Queue Manager control information followed by the specific data placed in the queue by the client.

Queue File Header

The queue file header contains all the information needed by the Queue Manager to manage the queue entries in the file. This information includes the following entries:

- The queue type, such as spooler or RJE
- The queue version
- A listing of all current queue servers
- Two sets of head and tail pointers to a doubly linked list of queue entries

As a consistency check, the Queue Manager matches the queue type against the type specified in all client and server operations.

The Queue Manager checks the queue version of existing queues against the Queue Manager version. Note that all queues created by an earlier version of the Queue Manager are interpreted correctly and thus can be managed by a later Queue Manager version. The reverse, however, is not true: queues created by a later Queue Manager cannot be used by an earlier Queue Manager version.

Two sets of head and tail pointers contain memory addresses in a doubly linked list of queue entries.

- One set contains the addresses of the first and last entries available for use.
- The other contains the addresses of the first and last entries currently being served or waiting to be served.

(For the format of the queue file header, see Table 12-2 under "Data Structures." later in this section.)

Queue Entry

A queue entry is a formatted request for processing that is added by clients to the specified queue. Clients and queue servers communicate by means of fields within the queue entries located at fixed offsets known to both the clients and the servers. When a queue server is available, it obtains a queue entry for processing.

Queue Entry Processing Order

Queue entries are priority ordered such that new entries are inserted after the last entry of higher priority, and before the first entry of lower priority. Priorities range from 0 through 9, with 0 the highest. A program can specify the queue entry priority with the AddQueueEntry operation.

Within a given priority, entries are arranged in a first-in, first-out (FIFO) order.

In addition to setting the priority of a queue entry, a program can use the AddQueueEntry operation to specify the earliest time at which to make the entry available for processing and a repeat time interval for making the entry available for processing at a later time. These options are represented by the *dateTime* and *repeatTime* parameters, respectively, of the AddQueueEntry operation.

The *dateTime* parameter can be used to specify that a queue entry not be made available for use until a particular date and time. For example, if today were February 15, 1993, a queue entry can be added to a queue that is first made available for use at 6:00 AM on March 2, 1993, and that subsequently remains available until its removal from the queue.

The *repeatTime* parameter can be used to specify a repeat time interval at which the queue entry again is made available for use. For example, a queue entry with a repeat time interval of 12 hours can be added to a queue that is first made available for use at 6:00 AM on March 2, 1993, and that is available twice every day thereafter until its removal from the queue.

Queue Entry Format

Each queue entry consists of the following two parts:

- The queue entry header is contained in the first 40 bytes. The header is reserved for the Queue Manager and includes control information for linking the queue entries together. (The format of the queue entry header is shown in Table 12-3 under "Data Structures" later in this section. The Queue Status Block, shown in Table 12-4, is derived from this header.)
- The remaining bytes are data placed in the queue entry by the client. This information rarely is used by the Queue Manager except in a few cases where queues are internally defined. (Tables 3-1 through 3-3 in Section 3, "Spooler," show this portion of the queue entry for pre-GPS spooler queue entries.)

Calculating Queue Entry Size

A queue entry is one or more contiguous 512-byte sectors in a queue. The smallest size for a queue entry is 512 bytes. Larger queue entries must be a multiple of 512 bytes.

The 16-bit links to the preceding and next queue entries in the queue entry header place a maximum limit on the number of queue entries in a queue. To determine the maximum number of queue entries, divide 65,534 by the queue entry size (in sectors). As an example, a queue with queue entries that are 4 sectors long can contain a maximum of (65534/4) or 16383 entries.

Queue Examples

More than one type of queue generally is required for each queue-oriented service. (A spooler queue, for example, requires a scheduling, a control, and a status queue.) Table 12-1 shows examples of typical queues.

Table 12-1. Queue Examples

Queue Server	Type	Number Required
Remote Job Entry (RJE)	Transmit	One per cluster configuration
	Receive	One per cluster configuration
Spooler	Scheduling	One per print class
	Control	One per printer
	Status	One per cluster configuration

Defining Queues

Queues can be defined by the system administrator before the Queue Manager is installed or after installation by programs that call the AddQueue operation.

Defining Queues in the Queue Index File

During Queue Manager installation, the Queue Manager obtains information about the queues that it is to manage by reading the contents of the file, *[Sys]<Sys>Queue.Index*. This file, also known as the queue index file, is located at the server or standalone workstation where the installation takes place.

The queue index file is a text file that defines queues to be used in the system. It contains information such as the name of each queue to be used in the system and the associated queue entry file.

Note: *The Queue Manager reads the queue index file only once (during Queue Manager installation). To activate any changes made to the file after installation, deinstall the Queue Manager and install it again.*

If required, the system administrator creates the queue index file in the *[Sys]<Sys>* directory at the server (or standalone workstation) where the Queue Manager is to be installed.

The queue index file is created with a text editor or word processor. For each queue, an entry of the following format is required:

QueueName / FileSpec / EntrySize / QueueType **<RETURN>**

where

QueueName

Is a user-defined queue name that is unique to the installation. The name can be any name of up to 50 characters. Following are examples of acceptable names: SpoolerA, SPL, PrinterX, Centronix, Diablo, Imagen2.0, and RJEtoBoston. Note that this name can be preceded by a node specification to specify that the queue itself is being serviced at a remote node by another Queue Manager. (For details, see "Using the Queue Manager Across Network Nodes," later in this section.)

FileSpec

Is the file specification of the queue in which queue entries submitted by clients are stored. Following is an example:

[Win1]<Sys>SpoolerAQueueEntryFile.

EntrySize

Is the size of a queue entry in the queue. The size is the number of 512-byte sectors per entry. For example, to define 1KByte entries, specify an entry size of 2. In such a case, 984 bytes are usable, and 40 are reserved for the Queue Manager. (Also see "Calculating Queue Entry Size," earlier in this section.)

QueueType

Is the type of the queue (an integer less than or equal to 255), which enables a consistency check. The Queue Manager checks the type against the type specified in operations to add entries to the queue and to establish servers for the queue. Types 0 through 80 are reserved for internal use. Types 1, 2, 3, and 4 are assigned as follows:

Type	Assignment
1	spooler queue
2	BSC 2780/3780 RJE queue
3	Batch queue
4	SNA RJE queue

An example of a queue index file is shown in Figure 12-2.

Figure 12-2. Example of a Queue Index File

```
[sys]<sys>SpoolerAQueueEntryFile/1/1 <Return>

[sys]<sys>RJEBostonQueueEntryFile/1/2 <Return>
```

606.12-2

Defining Queues Dynamically

Programs can add queues dynamically by calling the AddQueue operation and supplying the same information that is contained in the queue index file entry fields. The Queue Manager adds these queues to its tables and it creates, if necessary, and opens the specified queue entry file.

A queue handle returned by the AddQueue operation can be used by the caller in subsequent operations to reset the dynamic queue to its initial state, or to remove it entirely. (See the AddQueue, CleanQueue, and RemoveQueue operations in the *CTOS Procedural Interface Reference Manual*.)

Referencing Queues and Queue Entries

A program references a queue or a queue entry in a queue management operation in several ways.

With the exception of GetQmStatus, which requires that the caller supply only the queue type, a queue is referenced by the queue name or the queue handle as well as the type of the queue. Queue entries are referenced by the queue entry handle or keys embedded in the client portion (starting at byte 40) of the queue entry itself.

The four common methods of referencing queues and queue entries (by queue name, queue handle, queue entry handle, and keys) are described in detail in the following subsection.

Referencing Queues

A queue is identified by a name and a handle. Each queue management operation determines which type of queue identifier it needs to reference a queue.

Queue Names

Most queue management operations require that the caller supply the queue name. (Exceptions are when the queue is being removed or reset.)

If a queue is defined in the queue index file, the queue name (as well as the queue type) must be known in advance by programs using the queue. If, however, the queue is defined dynamically using the AddQueue operation, the program defining the queue actually supplies this information as part of the queue definition.

The only queue management operations that do not require a queue name are CleanQueue, RemoveQueue, DeinstallQueueManager, and GetQmStatus. The CleanQueue and RemoveQueue operations accept a queue handle. Calling GetQmStatus is one way this handle can be obtained, as described next.

Queue Handles

Queue handles are unique 16-bit identifiers of queues. These handles can be used by the queue manipulation operations, CleanQueue and RemoveQueue.

A program can obtain a queue handle in one of two ways. If the program defined the queue dynamically using AddQueue, a queue handle is returned. (See "Queue Manipulation Operations," earlier in this section.) If the queue is defined in the queue index file, the program must call GetQmStatus, specifying the queue type of the queue whose queue handle is to be obtained. Because GetQmStatus provides information about all queues of a specified type, the program must extract the queue handle from the returned information. (For a detailed description of GetQmStatus, see the *CTOS Procedural Interface Reference Manual.*)

Referencing Queue Entries

Referencing queue entries requires knowledge of queue entry handles, status blocks and keys, defined below.

Queue Entry Handles

Queue entry handles are unique 32-bit identifiers of queue entries. The caller must use the queue entry handle to specify to the Queue Manager which queue entry to remove, unmark, reschedule, or rewrite. A queue entry handle also is used by callers of the ReadNextQueueEntry operation to obtain the next queue entry.

Queue entry handles are returned by the following Queue Manager operations:

- MarkKeyedQueueEntry
- MarkNextQueueEntry
- ReadKeyedQueueEntry
- ReadNextQueueEntry

The handle is returned in the Queue Status Block structure.

Queue Status Block

The Queue Status Block is a structure derived from information contained in each queue entry header. This structure reports a queue entry's server user number (if the entry is marked), priority, and the buffers in which the queue entry handle for the queue entry, as well as for the logically following queue entry, are stored.

(For the structure of the Queue Status Block, see Table 12-4.)

Keys

A key is a particular field or combination of fields that identifies a queue entry to be processed. Unlike queue names, queue handles, and queue entry handles, which reference data in the Queue Manager control portion of the queue file header or a queue entry header, keys reference data actually stored within the client's portion of a queue entry. (See "Queue Entry Format," earlier in this section.) For this reason, keys are recognized by the Queue Manager only when they are defined by the caller.

The following operations allow a program to access queue entries by keys:

- ReadKeyedQueueEntry
- MarkKeyedQueueEntry
- RemoveKeyedQueueEntry

Note that the MarkKeyedQueueEntry operation can be used only by a program established as a queue server. (For details, see "Queue Server Operations," earlier in this section.)

In the operations listed above, a program specifies one or two queue entry key fields and their offsets within the queue entry. The Queue Manager uses this information to search the queue until it finds either the first entry containing the specified key(s) or no entry containing the specified key.

Each key field in the queue entry is described with an sb string: the first byte contains the key field length and the remaining bytes contain the key.

In the following example, each entry consists of three key fields (Date/Time, Title, and Message). The format of the entry is as follows:

Offset	Field	Size (bytes)	Description
0	sDateTime	1	Length of date/time specification
1	DateTime	4	Date/time specification
5	sTitle	1	Length of a title
6	Title	20	Title buffer
26	sMessage	1	Length of a message
27	Message	255	Message buffer

For examples, a program can delete the first queue entry in the queue with a key field containing the following Title:

From Mom

It calls RemoveKeyedQueueEntry and specifies the title 'From Mom' as the key, the key length (8 bytes), and the key offset (5).

RemoveKeyedQueueEntry also can be called to delete the first queue entry containing a specified title as well as a date/time specification. In this case, a second key containing the date/time specification, its length (4 bytes), and its offset (0) also is used in the search.

Note: *Any program using keys must know the exact format and content of queue entries in the target queue entry file. It follows that multiple programs accessing the same queue must agree at least upon the contents of queue entries in that queue.*

Sequence for Using Queue Management Operations

A typical sequence for installing and using the queue management facility is described below.

1. The Queue Manager is installed on the server or the standalone workstation with a Batch utility, the Executive **Install Queue Manager** command, or the Print Manager. At installation, the system administrator can choose to specify a maximum number of queues that can be defined dynamically.
2. If a queue index file exists, the Queue Manager uses it during initialization to open the queues defined within the file. (Otherwise, it installs with the dynamic queues, if specified.)
3. At any time after the Queue Manager is installed, programs can add queues dynamically with the AddQueue operation, provided that dynamically defined queues were specified when the Queue Manager was installed. (See step 1.) The number of queues defined in this manner must not exceed the maximum number specified at Queue Manager installation. The Queue Manager adds these queues to its tables, and it creates, if necessary, and opens the specified queue entry file.
4. A program intending to serve a particular queue uses the EstablishQueueServer operation to establish itself as an active queue server. It specifies the name of the queue it intends to serve and the queue type.
5. A client adds queue entries to the specified queue with the AddQueueEntry operation. The *fQueueIfNoServer* parameter to AddQueueEntry allows the client to indicate whether a queue server is established for the queue. In either case, the Queue Manager places the entry in the queue.
6. The queue server can obtain a particular queue entry for processing with the MarkKeyedQueueEntry operation. It also can obtain the unmarked queue entry at the head of the queue by using MarkNextQueueEntry. The Queue Manager marks the queue entry as being in use to prevent other queue servers from operating on it.

7. The queue server processes the marked queue entry and then removes the entry from the queue using the RemoveMarkedQueueEntry operation. The server also can either reschedule the queue entry (if a repeating time interval is specified for the queue entry), or simply unmark it, making it available for further processing at a later time.
8. To obtain a list of entries in the queue, the client can call ReadNextQueueEntry repeatedly. The client also can call RemoveKeyedQueueEntry to delete an entry before the entry is marked by the queue server.
9. When a queue server elects to discontinue serving a queue, the queue server removes itself from the list of active queue servers by calling the TerminateQueueServer operation.

Using the Queue Manager Across Network Nodes

Programs can access queues residing on remote network nodes. Remote access is transparent to the calling program that uses the following subset of the queue management operations:

- AddQueueEntry
- ReadKeyedQueueEntry
- ReadNextQueueEntry
- RemoveKeyedQueueEntry

Calling any other queue management operation, however, results in the return of status code 920 (not a remote Queue Manager operation).

To access a remote queue, the queue must be defined in the local queue index file, as well as in the queue index file at the remote node. (See "Defining Queues in the queue index File," earlier in this section.) The only difference in the local and remote queue definition is that the local queue name must include the node name of the remote node. Installation of the Queue Manager at both locations then allows programs to use the operations listed above.

The local Queue Manager forwards requests with node specifications to the Queue Manager at the remote node specified in the queue name.

The following steps describe how to define a remote queue:

1. At the local node, create an entry in the queue index file. Enter the node specification of the queue preceding the queue name. Note that since the Queue Manager at this node will forward any requests to use this queue, the remaining elements of the entry (queue file specification, queue type, and queue size) are not used, but must be present for Queue Manager installation to succeed.
2. At the remote node, set up an entry in the queue index file using the same queue name, but omitting the node name. Omission of the node means that the queue is to be served locally (at this node). Include the remaining elements of the entry using the same queue file specification, queue type, and queue size as were specified in step 1.
3. Install the Queue Manager at the local and remote nodes.

Once the Queue Managers are installed, they are provided the required information for recognizing and processing requests across the network. Subsequent use of a remote queue is totally transparent to programs using the subset of queue management operations listed earlier.

Following is an example of the local and remote queue index file entries:

At the server at the local node, the queue index file contains the following entry:

```
{Ranger}RmtQueue/[d0]<Queues>Rmt.Queue/1/8
```

This local entry requires that the queue index file at the server at the remote node {Ranger}, contain the corresponding entry:

```
RmtQueue/[d0]<Queues>Rmt.Queue/1/8
```

Although the information in the local queue index file following the queue name (/[d0]...1/18) is unused by the local Queue Manager, it must be present and must match the information in the queue index file at the remote node.

Including a node name in the queue file specification is not recommended. By doing so, the Queue Manager servicing the queue is required to perform multiple reads and writes of data across the network, even to perform a single Queue Manager function. This could lead to impaired system performance.

Data Structures

This subsection contains data structures important to queues. Table 2-2 provides information about queue files header; listed field are described below the table.

Table 12-2. Queue File Header

Offset	Field	Size (bytes)	Description
0	*version*	4	Current version string for the queue
4	*queueType*	2	Value from 1 to 255
6	*fUnique*	1	Queue access by only one queue server
7	*cClients*	1	Number of queue servers for this queue
8	*freeTop*	2	First free block to hold a new queue entry
10	*queueTop*	2	Block index of the first queued entry
12	*freeBot*	2	Last free block to hold a new queue entry
14	*queueBot*	2	Block index of the last queued entry
16	*priTops*	20	Linked list of one-word block indexes
36	*priNext*	20	Linked list of one-word block indexes
56	*reserved*	4	
60	*rqServers*	128	Queue server user numbers for this queue
188	*rqQdMarkRq*	256	Queue servers waiting for queue entries
444	*earliestDT*	4	Earliest time for delayed/repeating entry
448	*earliestId*	2	Queue entry with the time stamp earliestDT.
450	*latestDT*	4	Latest time for delayed/repeating entry
454	*latestId*	2	Queue entry with the time stamp latestDT.
456	*fStable*	1	Queue file properly closed when last used
457	*cQdEntries*	2	Is the number of queue entries
459	reserved	54	

version

Is the current version string for the queue (for example, 11.3).

queueType

Is a value from 1 to 255. Types 1, 2, 3, and 4 are assigned as follows:

Type	Assignment
1	spooler queue
2	BSC 2780/3780 RJE queue
3	Batch queue
4	SNA RJE queue

fUnique

Is a flag that is TRUE if only one queue server is allowed to access the queue.

cClients

Is the number of programs currently established as queue servers for this queue.

freeTop

Is the sector index of the first free sector (available to contain a new queue entry).

queueTop

Is the sector index of the first queued entry.

freeBot

Is the sector index of the last free sector (available to contain a new queue entry).

queueBot

Is the sector index of the last queued entry.

priTops

Is a linked list of sector indexes (one word each) of the first queue entry for a given priority. (The first word contains the first sector index with a priority 0 entry, if any. The last word contains the first sector index with a priority 9 entry.) Priorities for which there are no entries contain the value 0.

priNext

Is a linked list of sector indexes (one word each) of the first queue entry with the next highest priority.

rqServers

Is an array of user numbers of programs currently established as queue servers for this queue.

rqQdMarkRq

Is an array of user numbers of queue servers waiting for available queue entries.

earliestDT

Is the earliest time stamp for any delayed or repeating queue entry in the queue.

earliestId

Is the sector index of the queue entry with the time stamp, earliestDT.

latestDT

Denotes the latest time for any delayed or repeating queue entry in the queue.

latestId

Is the sector index of the queue entry with the time stamp, latestDT.

fStable

Is a flag that is TRUE if the queue file was properly closed when it was last used.

cQdEntries

Is the number of entries in the queue.

Table 2-3 provides information about queue entry header; listed field are described below the table.

Table 12-3. Queue Entry Header

Offset	Field	Size (bytes)	Description
0	*idxSelf*	2	Block index of this queue entry
2	*uniqId*	2	Unique ID for this queue entry
4	*chainType*	1	Link to list of queued/available entries
5	*priority*	1	Priority (0 to 9) of this queue entry
6	*nxtUniqId*	2	Unique ID of next queue entry in the queue
8	*prevIdx*	2	Block index of the preceding queue entry
10	*nextIdx*	2	Block index of the next queue entry
12	*dateTime*	4	Indicates when queue entry is to be used
16	*repeatInt*	2	Repeat interval for rescheduling entry
18	*markUNum*	2	User number of current queue server
20	*prevDTIdx*	2	Block index of the next earlier queue entry
22	*nextDTIdx*	2	Block index of the next later queue entry
24	reserved	16	

idxSelf

Is the sector index of this queue entry.

uniqId

Is a unique identification number for this queue entry. *idxSelf* (above) and uniqId are the queue entry handle for this entry.

chainType

Indicates whether this entry is linked to the list of queued or available entries.

priority

Is the priority (0 to 9 with 0 the highest) of this queue entry.

nxtUniqId

Is a unique identification number of the next queue entry in the queue.

prevIdx

Is the sector index of the preceding queue entry in the queue.

nextIdx

Is the sector index of the next queue entry in the queue. *nxtUniqId* (above) and nextIdx are the queue entry handle of the next entry.

dateTime

Is a time stamp used to determine when this queue entry is to be made available for use.

repeatInt

Is the repeat interval to be used in rescheduling this queue entry.

markUNum

Is the user number of the queue server that currently is using (has marked) this queue entry.

prevDTIdx

Is the sector index of the queue entry with the next earlier time stamp.

nextDTIdx

Is the sector index of the queue entry with the next later time stamp.

Table 2-3 provides information about queue status block; listed field are described below the table.

Table 12-4. Queue Status Block

Offset	Field	Size (bytes)	Description
0	*qehRet*	4	buffer for queue entry handle
4	*priority*	1	priority of the queue entry
5	*serverUserNum*	2	user number of the current queue server
7	*qehNextRet*	4	buffer for next queue entry handle

qehRet

Is the buffer in which the queue entry handle of the queue entry is stored.

priority

Is the priority (0 to 9, with 0 the highest) at which the queue entry is placed in the queue.

serverUserNum

Is the user number of the queue server that currently is using (has marked) this queue entry. This field contains 0FFFFh if the queue entry currently is not marked.

qehNextRet

Is the buffer in which the queue entry handle of the logically following queue entry is stored.

Section 13
Spooler

Introduction to Spooler

The spooler (simultaneous peripheral operation online) management facility provides spooled printing to parallel (Centronics-compatible) and serial (RS-232-C-compatible) printer interfaces.

In *spooled printing*, a queue entry is created for each printing request and is entered in a queue managed by the Queue Manager. (For details on the Queue Manager, see Section 2, "Queue Manager.") A spooler obtains a queue entry for printing when a printer is available. The user need not wait for a printer to be available to enter a printing request.

The user can print with the *Spooler Status* command, described in the *CTOS Executive Reference Manual.* The reader should be familiar with the pertinent text in that manual before continuing in this section.

Spooler Configuration

When the spooler is installed, it reads a spooler configuration file designated by the user. The spooler configuration file at spooler installation must contain at least the specification of each printer channel to be controlled by the spooler. Additional information required for each printer can be supplied to the spooler in either of two ways:

- In the spooler configuration file at spooler installation
- Dynamically through the ConfigureSpooler operation

The additional information required for each printer is

- The name of the printer
- The name of the scheduling queue
- The printer configuration file specification
- The priority of the process that controls the printer
- Whether to print a banner page at the beginning of each file

See the *CTOS Generic Print System Administration Guide* for information on setting up and configuring the spooler.

Sending a Password

If the security mode is specified in a printing request, the spooler pauses before printing the file and waits for receipt of a password. The password can be sent to the spooler in either of two ways:

- By the operator invoking the **Spooler Status** command and typing the password at the local printer
- By a process using the SpoolerPassword operation

Operations

Spooler management provides the operations described below.

ConfigureSpooler

Sets or changes the spooler's configuration.

SpoolerPassword

Sends a file password to the spooler.

Programmer's Notes on the Spooler

The information below is provided for programmers writing application programs that use the spooler.

Pre-GPS Spooler Byte Streams

Spooled printing can be accessed through pre-GPS spooler byte streams. (See "Sequential Access Method," in the *CTOS Operating System Concepts Manual*, for a description of pre-GPS spooler byte streams.)

The queue name must be enclosed in brackets (for example, [Joe]) to distinguish it from a file specification. The name has a maximum length of 12 characters and must not match a built-in byte stream device. (See "Sequential Access Method," in the *CTOS Operating System Concepts Manual*, for details on device/file specifications.)

During the OpenByteStream operation, the pre-GPS spooler byte stream creates a temporary file whose name is automatically generated to ensure its uniqueness. A WriteBsRecord or WriteByte operation transfers text to the temporary disk file and expands the disk file as necessary. The CloseByteStream operation closes the disk file and then sends the file to the specified scheduling queue. (For descriptions of the OpenByteStream, WriteBsRecord, WriteByte, and CloseByteStream operations, see the *CTOS Procedural Interface Reference Manual.*)

In a cluster environment, the temporary file is created at the server. In the operating system configuration file, the system administrator can specify the volume on which the temporary files are to be created. The byte stream specifies *[!Scr]* as the volume name of the temporary file, and the file system replaces *[!Scr]* with the actual volume name. The symbol ! is interpreted by the local file system to specify a volume at the server. (For details, see "File Management," in the *CTOS Operating System Concepts Manual.*)

In addition to the queue name, the user can optionally specify a document name following the queue name ([Joe]Report). The pre-GPS spooler byte stream creates the temporary file called *[!Scr]<Spl>Report$$xxxxx*, where *$$xxxxx* is a unique sequence generated by the byte stream. The installation procedure creates directory *<Spl>* on the volume *[!Scr].*

The pre-GPS spooler byte stream creates temporary files without password protection and deletes them after printing. If the security of temporary files is a concern, the user should create a disk file with the byte stream, and then spool it with the **Print** command, using the security mode. (See the *CTOS Executive Reference Manual.*)

Spooled printing can be reconfigured through the ConfigureSpooler operation. (See the *CTOS Procedural Interface Reference Manual.*)

Spooler Configuration File Requirements

During installation, the spooler reads a Spooler Configuration file designated by the user.

When the spooler is installed, the Spooler Configuration file must contain at least the predefined code of each printer channel to be controlled by the spooler (even if the printer is not configured at this time). The code(s) tells the spooler how much memory space to allocate for printers.

Additional information required for each printer can be specified by either of the following:

- Including it in the spooler configuration file when the printer spooler is installed
- Using the ConfigureSpooler operation during execution or the **Spooler Status** command

The additional information required for each printer is the following:

- The name of the printer
- The name of the scheduling queue
- The printer configuration file specification
- The priority of the printer control process (See "Control Queue," later in this section)
- A code to indicate whether a banner page is printed between files

Printer Spooler Escape Sequences

Printer spooler escape sequences are special character sequences embedded in text files to be printed by the printer spooler. They either cause an intentional manual intervention condition when processed by the printer spooler or override the page count generated by the printer spooler. The format for a printer spooler escape sequence is

0FFh, *type, cbText, text*

where

type

Identifies the reason a manual intervention is required:

Value	Description
1	forms change
2	print wheel change
3	generic printer pause
4	page number overwrite

cbText

Is the count of bytes in the following text. The maximum is 12 for types 1 and 2, and 60 for type 3.

text

Is a character string that identifies the chosen form or print wheel, the reason for the generic printer pause, or the page number.

Queue Management

Traditional spooler queues are type 1 queues, which require that the Queue Manager verify the entries added to the queue. The runtime determination of the printer queues is achieved with the GetQMStatus operation. GetQMStatus is used to specify the queue type (in the case of the spooler, type 1) and the number of spooler queues, along with each queue name and its associated queue handle.

The spooler queues returned are of the following classes:

- Scheduling (for example, SPL)
- Status (for example, SpoolerStatus)
- Control (for example, SPLControl)

Scheduling Queue

If an application program is reading through a scheduling queue with the ReadNextQueueEntry operation (described in Section 2, "Queue Manager"), it can determine whether an entry is currently being printed by looking at the *serverUserNum* field of the returned Queue Status Block. If the *serverUserNum* field is 0FFFFh, the entry is waiting to be printed; otherwise, the entry is currently printing.

Table 13-1, under "Data Structures," later in this section, shows the format of a scheduling queue entry.

Status Queue

If an application program is reading through a status queue with the ReadNextQueueEntry operation, the only active (valid) printer status entries are those that are marked. A program can determine which entries are marked by looking at the *serverUserNum* field of the returned Queue Status Block. If the *serverUserNum* field is 0FFFFFh, the entry is not an active printer.

Table 13-2, under "Data Structures," later in this section, shows the format of a status queue entry.

Control Queue

The following operations use the control queue:

- Align Printer
- Cancel Printer
- Halt Printer
- Restart Printer

The control queues have a specific queue entry format:

Command	byte
Restart Page	word
sbWpRestartPage (13)	byte

To perform these functions, an AddQueueEntry operation for a specific control queue is specified with the desired command entered in the queue entry.

Table 13-3, under "Data Structures," later in this section, shows the formt of a control queue entry.

ConfigureSpooler Program Example

To remove a printer channel from the spooler's control, the following operations are required:

1. This step enables you to determine whether the printer is attached locally. You can remove a printer channel only if it is attached locally.

   Call ConfigureSpooler with the channel specified and null values for all other parameters. If an erc 702 results (invalid printer), the printer is under the spooler's control. If an erc 33 results (service not available), the printer is not attached locally and you will not be able to remove it.

2. If the printer is local, follow this step to remove the printer channel. Call ReadKeyedQueueEntry on the SpoolerStatus queue with the specified printer. If an erc 0 results, the channel is free. If any other error is reported, a problem exists and the printer channel has not been removed.

SpoolerPassword Program Example

If a print request through the Executive **Print** command has been issued and the [*Security Mode?*] field has a Yes value, the file will be queued, waiting for printing, until a SpoolerPassword request is issued. The status queue for this printer indicates a paused state (1). If the scheduling queue entry shows that a password is required (*fSecurityMode* = TRUE), the entry is expecting a password. The SpoolerPassword request is then issued to allow printing to continue.

Data Structures

This section describes data structures relevant to spooler programming. Table 13-1 shows information on the spooler scheduling queue entry; the fields are described after the table.

Table 13-1. Spooler Scheduling Queue Entry

Offset	Field	Size (bytes)	Description
0	*fDelAftPrt*	1	TRUE deletes spooled file after printing
1	*sbFileSpec*	92	Name of the file to be printed
93	*sbFormName*	13	Name of the form to be used
106	*sbWheelName*	13	Name of the print wheel to be used
119	*cCopies*	2	Number of copies of the file to be printed
121	*bPrintMode*	1	Printing mode
122	*fAlignForms*	1	TRUE uses the forms alignment option
123	*fSecurityMode*	1	TRUE prints the file in security mode
124	reserved	5	
129	*sbDocName*	92	Name of the document being printed
221	*sbUserName*	31	Client's user name
252	reserved	4	
256	*timeQueued*	4	Date and time that the print was queued
260	*fSupressNewPage*	1	TRUE supresses form-feed at start of print
261	*fWPPaging*	1	TRUE uses WP page escape sequences
262	*fSupressBanner*	1	TRUE suppresses banner on the notice file
263	*fSingleSheet*	1	TRUE means printer is manual feed
264	reserved	20	

fDelAftPrt

Is a flag. TRUE deletes the spooled file after it is printed.

sbFileSpec

Is the name of the file to be printed. The first byte of the sb string is the string length.

sbFormName

Is the name of the form to be used. If the length is 0, the standard form is used.

sbWheelName

Is the name of the print wheel to be used. If the length is 0, the standard print wheel is used.

cCopies

Is the number of copies of the file that are to be printed.

bPrintMode

Is the printing mode. The values are

Mode	Description
0	Normal
1	Image
2	Binary

fAlignForms

Is a flag. TRUE means the forms alignment option is to be used.

fSecurityMode

Is a flag. TRUE means the file is to be printed in security mode with password protection.

sbDocName

Is the name of the document being printed. This is different from *sbFileSpec*, which is typically a temporary file in the *[!Scr]<Spl>* directory.

sbUserName

Is the client's user name.

timeQueued

Are the date and time that the print was queued.

fSupressNewPage

Is a flag. TRUE means that a form-feed is not performed at the start of the print job.

fWPPaging

Is a flag. TRUE means that WP page escape sequences will be to determine page numbers.

fSupressBanner

Is a flag. TRUE means that a banner will not be printed on the notice file.

fSingleSheet

Is a flag. TRUE means the printer attached is manual feed.

Table 13-2 shows information on the spooler status queue entry; the fields are described after the table.

Table 13-2. Spooler Status Queue Entry

Offset	Field	Size (bytes)	Description
0	*sbPrinterName*	13	Name of the printer
13	*sbCurrentPage*	13	Character sequence defining page number
26	reserved	25	
51	*sbQueueName*	51	Name of the queue the printer is serving
102	*bChannelNum*	1	Channel used
103	*sbConfigFile*	79	Printer configuration file name
182	*fAtServer*	1	TRUE if the systen service is at the server
183	*bStatus*	1	Printer status
184	*sbSpooledFile*	79	Name of the currently printing file
263	*sbWheelName*	13	Name of the current print wheel
276	*sbFormName*	13	Name of the current forms
289	*sbPauseMessage*	61	Pause message to be displayed
350	*fNeedWheelChange*	1	TRUE if a different print wheel is needed
351	*fNeedFormsChange*	1	TRUE if a different form is needed
352	*fShowPauseMsg*	1	TRUE means display the pause message
353	*wsNum*	2	Workstation number
355	reserved	2	
357	*sbDocName*	79	Name of the document being printed
436	*sbUserName*	31	Client's name
467	*timeStarted*	4	Date and time that the print was started

sbPrinterName

Is the name of the printer.

sbCurrentPage

Is the character sequence that defines a page number in a word processor print file.

sbQueueName

Is the name of the queue the printer is serving.

bChannelNum

Is the channel used. See the *CTOS Operating System Concepts Manual* for a description of the channels.

sbConfigFile

Is the name of the printer configuration file.

fAtServer

Is a flag. TRUE means the system service is located at the server of a cluster.

bStatus

Is the printer status. The status values are

Value	Description
0	Idle
1	Paused
2	Printing
3	Offline
4	Down

sbSpooledFile

Is the name of the currently printing file.

sbWheelName

Is the name of the current print wheel. If the length is 0, the standard print wheel is being used.

sbFormName

Is the name of the current forms. If the length is 0, the standard forms are being used.

sbPauseMessage

Is the pause message to be displayed.

fNeedWheelChange

Is a flag. TRUE means a different print wheel is needed.

fNeedFormsChange

Is a flag. TRUE means a different form is needed.

fShowPauseMsg

Is a flag. TRUE means the pause message should be displayed.

wsNum

Is the workstation number.

sbDocName

Is the name of the document being printed.

sbUserName

Is the client's name.

timeStarted

Are the date and time that the print was started.

Table 13-3 shows information on the spooler control queue entry; the fields are described after the table.

Table 13-3. Spooler Control Queue Entry

Offset	Field	Size (bytes)	Description
0	bCommand	1	command to the spooler
1	restartPage	2	restart printing from this page number
3	sbWpRestartPage	13	description of page from which to restart

bCommand

is the command to the spooler. The command values are

Value	Description
0	Halt/pause printer
1	Cancel print
2	Restart printer
3	Align forms

restartPage

Is the page number from which to restart printing. If this value is 0, the printing restarts at the beginning of the current page. If this value is 0FFFFh, the printing starts at the next character in the file.

sbWpRestartPage

Is the name describing the page from which to restart printing (for example, a Roman numeral). If the first byte in this entry is 0, this field is ignored.

Section 14
Voice/Data Services

Overview

The Voice/Data Services provide the Telephone Service on B25 X-Bus workstations. The Telephone Service provides an interface between application programs and the Voice Processor Module. All references to X-Bus workstations in this section include the following workstation types:

- B39/Series 386i
- B38/Series 386
- B28/Series 286

To use the operations described in this section, first use the **Install Voice Service** command to install the Telephone Service as described in the *CTOS Executive Reference Manual.*

Note: *Voice Services are supported only on CTOS I and CTOS II.*

The Telephone Service, the system service that manages Voice/Data Services on a X-Bus workstation, is an extension of your current operating system. Application programs use the Telephone Service request and procedural interfaces to access the Voice Processor hardware for voice, telephone, and data operations.

Using the Telephone Service, an application program can record and play back voice messages, manage telephone functions, and transmit data over a modem. One example of an application using the Telephone Service would be a computerized system in which a customer telephones the bank, listens to a recorded message, and then enters digits on the telephone keypad to complete a series of banking transactions. Other simple applications include telephone answering machine packages, dialing programs, and data transfer applications.

Another important component of the Telephone Service is the Telephone Status monitor program (described below). The Telephone Status monitor program is an X-Bus debugging tool used primarily by programmers.

Telephone Management

The telephone management facility of the Telephone Service is supported on X-Bus workstations only. This facility allows an application program to use all the telephone functions of the Voice Processor hardware, including automatic dialing, placing a call on hold, switching between calls on different telephone lines, and automatic answering of a ringing telephone line.

Telephone Status Monitor Program

The Telephone Status monitor program provides a useful debugging tool for software developers on X-Bus workstations. The **Telephone Status** command allows the programmer to view a simplified picture of the hardware circuits that are connected when a telephone management function is performed. The Status Monitor has function keys that allow the programmer to select which lines to place on hold, which lines to hang up, and which lines to link for conference calling. As each function is performed, the screen provides a visual representation of the circuits connected for the operation.

Data Management

The data management facility of the Telephone Service is supported on X-Bus workstations only. This facility allows an application program to use the optional internal modem of the Voice Processor Module. Data transfers can be made from a local workstation or from a server on the network. User interaction with the modem can be direct, via the Asynchronous Terminal Emulator (ATE), or transparent, via an application program such as the Operator or electronic mail.

Functional Groups of Operations

The following sections offer a brief description of the Voice/Data Services operations. See the *CTOS Procedural Interface Reference Manual* for complete descriptions of these operations. With the Telephone Service, the application and the service must run on the same workstation, except that data management requests of the Telephone Service can be routed across the network.

Voice Management Operations

TsVoiceConnect

Specifies the connection to make for a subsequent TsVoicePlaybackFromFile or TsVoiceRecordToFile operation when the *fAutoStart* flag in the Voice Control Structure is FALSE. (See Table 14-7 for the Voice Control Structure.)

TsVoicePlaybackFromFile

Plays back voice from the specified file (or from memory) to the output device. The output device is the telephone handset or one of the telephone lines.

TsVoiceRecordToFile

Digitizes voice from the input source to the specified file. The input source is the telephone handset or one of the telephone lines.

TsVoiceStop

Terminates the recording or playback of a voice message.

Telephone Management Operations

TsConnect

Connects and disconnects the telephone unit and the telephone lines.

TsDeinstall

Deinstalls the Telephone Service.

TsDial

Causes the dual-tone multifrequency (DTMF) encoder to generate specified characters. (See Table 14-1.)

TsDoFunction

Performs functions similar to those available on a two-line telephone unit, such as selecting a telephone line, placing it on hold, hanging up, turning on the speaker phone, and so on. This function also locks and unlocks the voice encoder/decoder (CODEC), which allows certain parts of the hardware and Voice/Data Services to be reset as well as providing Context Manager switching.

TsGetStatus

Returns the status of the Voice Processor Module and of all users. (See Table 14-4.)

TsHold

Places the specified telephone line on hold.

TsLoadCallProgressTones

sets the call progress tones to be used for a subsequent TsDial request. This call is necessary only when the call progress tone values are to be changed.

TsOffHook

places the specified telephone line offhook.

TsOnHook

places the specified telephone line onhook (that is, hangs it up).

TsQueryConfigParams

returns the Telephone Service configuration file name and the current configuration information (see Table 14-3, Telephone Service Configuration Block).

TsReadTouchTone

reads DTMF characters from one of the telephone lines or from the telephone unit.

TsRing

turns on (or off) the video monitor ringing. The ringing is turned on by specifying a video monitor ring frequency between 1 and 255. Video monitor ringing is turned off by specifying a frequency of 0. This operation allows an application and user to select a ring frequency interactively.

TsSetConfigParams

sets configuration information for the Telephone Service (see Table 14-3, Telephone Service Configuration Block).

TsVersion

returns the version of the Telephone Service.

Data Management Operations

TsDataChangeParams

Allows a program to change the parity, line control, and other parameters for an open line.

TsDataCheckpoint

Causes a data line to be checkpointed. The request returns only after all TsDataWrite requests have been returned and after the modem has transmitted the last character.

TsDataCloseLine

Closes a data line and terminates the call.

TsDataOpenLine

Starts a data session using the modem.

TsDataRead

Moves a block of received data from the modem to the specified user memory.

TsDataRetrieveParams

Returns the parameters of the specified line.

TsDataUnAcceptCall

Allows a program to retrieve a TsDataOpenLine request before the timeout expires.

TsDataWrite

Writes a block of data to the modem.

Hardware Features of the X-Bus Workstation

This section describes the major hardware features of the X-Bus workstation that pertain to the Voice/Data Services.

Voice Features

Important hardware features of the X-Bus workstation that relate to voice management are the CODEC and the voice amplifier, described in the following sections.

CODEC

A speech analyzer/synthesizer, known as the CODEC (encoder/decoder) converts analog (voice) signals to digital (binary) signals and back again. This feature allows your workstation to store and retrieve voice messages as file data. The answering-machine functions of some application software use the CODEC.

The CODEC is not shareable; that is, it is not possible to play back or record on one telephone line while playing back or recording on the other line.

Voice Amplifier

The Voice Processor Module includes a voice amplifier that can be used when signal strength is low (such as from a speaker phone) to improve recording quality.

Telephone Features

This section provides a brief overview of the X-Bus Voice Processor hardware features used by the Telephone Service. These features include the telephone lines, DTMF tone generator and receiver, and call progress tone detector. For more detailed information, see the manual for your voice processor.

Two Telephone Lines

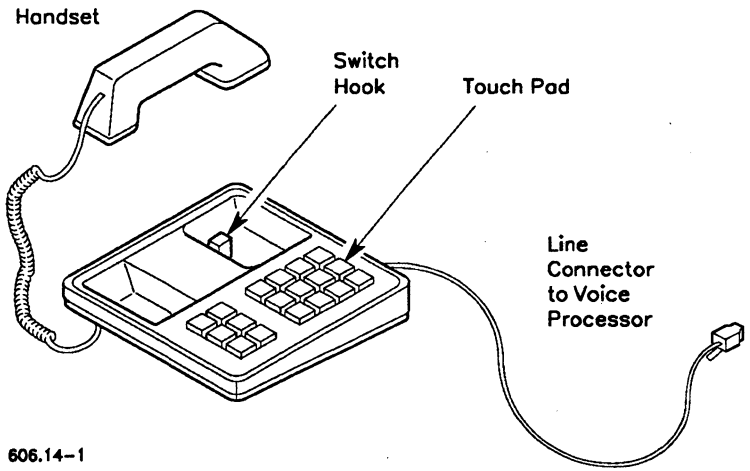
The Voice Processor Module supports two telephone lines. A standard *telephone unit* (see Figure 14-1) is connected to the "PHONE" connector on the rear of the Voice Processor Module. Separate telephone cables connect each of the two Voice Processor *telephone lines* to the telephone wall jacks, as shown in Figure 14-2. Although it is possible to have two telephone lines, only one is required.

The Voice Processor Module enhances operation of the telephone unit but does not replace it. If your workstation is turned off, the telephone reverts to normal operation and defaults to line 1 if you have two telephone lines.

DTMF Tone Generator and Receiver

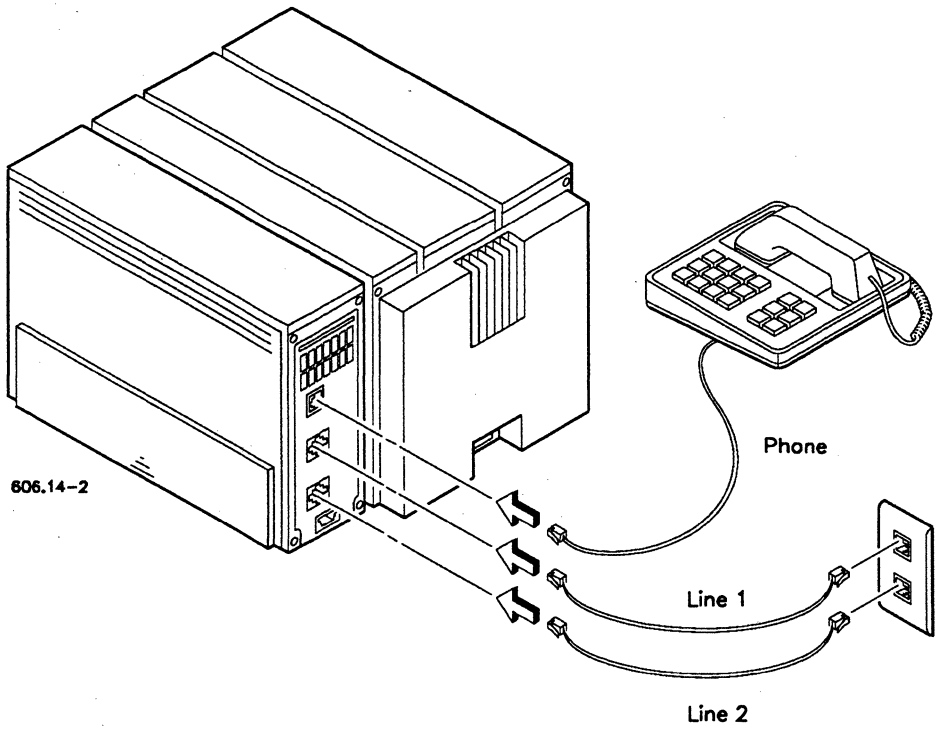
The dual-tone multifrequency (DTMF) touch-tone autodialer is a standard feature of the Voice Processor Module. The DTMF *tone generator* produces all sixteen DTMF digits (0 through 9, *, #, and A through D) under software control and is normally used in autodial applications. The DTMF tone generator can be programmed to transmit various key sequences to activate specific features of private branch exchange (PBX) systems. It can also produce single tones. See "Dialing," later in this section, for more information on how DTMF tones are generated.

Figure 14-1. Parts of the Telephone Unit



The *DTMF tone receiver* decodes incoming DTMF tones as digits for use by application programs. For example, numeric information can be input using the telephone unit touch pad and used by programs supporting voice mail or voice response capabilities. The Voice Response System program example, later in this section, uses this hardware feature.

Figure 14-2. Voice Processor Module Connections



Call Progress Tone Detector (CPTD)

The call progress tone detector (CPTD) consists of analog circuitry that, in conjunction with the Telephone Service, detects a busy signal, dial tone, reorder tone (fast busy), and answer (ringback) tone.

Data Features

Important hardware features of the X-Bus workstation that relate to data management are the modem and the analog crosspoint switch array, described in the following sections.

Modem

Some models of the Voice Processor Module include an internal modem. The modem can be used in both originate and answer modes and supports full-duplex transmission (asynchronous mode) over ordinary two-wire telephone circuits. This modem is compatible with the Western Electric 212A series at 1200 baud and the Bell 103/113 series at 300 baud.

Analog Crosspoint Switch Array

All Voice Processor devices are connected to an analog crosspoint switch array. Under software control, this switch array allows either telephone line to be connected to the telephone unit, the modem, the CODEC, the DTMF generator, or the DTMF decoder. See also the **Telephone Status** command, described later in this section, for a graphic representation of the analog crosspoint switch array.

Software Concepts

The following sections describe key software concepts as they relate to voice recording, telephony, and data.

Voice Recording

The voice interface is a set of requests that allow application programs to use the CODEC for voice annotation of files, for voice messaging, or for use with a software answering machine. The following paragraphs describe key concepts involved in writing programs that use the voice interface.

Recording Rates

On X-Bus workstations, the CODEC operates at either a 6kHz or an 8kHz sampling rate, generating digitized voice information at either 24Kbps (for 6kHz) or 32 Kbps (for 8kHz). Use the 8kHz rate for high-quality voice transmission. Use the 6kHz rate for applications where storage and transmission costs are a greater concern than voice quality. The *f6kHz* flag in the Voice Control Structure (see Table 14-7) specifies the recording rate.

Pause Compression

Pause compression can be used to reduce the size of voice files. During recording, when pause compression is enabled, low threshold sound is replaced with pure silence for a particular period. The indication of silence is recorded as escape sequences 7Fh-0F7h, as described in detail in the following paragraph. X-Bus workstations support pause compression on both record and playback.

During Recording. To enable pause compression, set the *fNoPause* and *fRawData* flags in the Voice Control Structure (Table 14-7) to FALSE. When these flags are set to FALSE and the sound level falls below a certain threshold, the byte 7Fh replaces the current sample to mark the beginning of the low sound period. (The actual CODEC data is still recorded during this low-sound period.) When the voice level rises above the minimum threshold, the escape character 0F7h is added to the data stream to terminate the low-sound sequence.

When pause compression is enabled as described above, the Telephone Service interprets the escape sequences before the file is written to disk. When it encounters a 7F escape byte, it removes all the data bytes up to the F7 escape byte and replaces them with a count of the bytes read.

Playback of Compressed Files. During playback of compressed files, set the *fRawData* flag to FALSE. (The *fNoPause* flag has no effect during playback.) The pause compression escape sequences are then expanded to their original length when the voice file is played back.

Note that if *fRawData* is set to TRUE during playback, pause compression escape sequences in the voice file (if any) are not be expanded; they are simply "played." This may result in slight anomalies in the sound output, most notably the running together of words.

Advantages and Disadvantages. Use of pause compression produces smaller files, but the voice file, when played back, may sound somewhat choppy because periods of low sound have been compressed and then expanded. Noncompressed recording of voice files results in slightly higher quality voice recordings, at the expense of disk space (approximately one-third more).

Amplification

When a voice message is being recorded on an X-Bus workstation, you can specify the use of an alternate connection that automatically increases the volume. To specify this amplified connection, set the *fAltConnection* flag in the Voice Control Structure to TRUE. (See Table 14-7.) There is no way to amplify playback on an X-Bus workstation.

Memory and Disk Files

The disk requirement without pause compression is 4000 bytes/second at 8kHz and 3000 bytes/second at 6kHz. With pause compression on X-Bus workstations, the disk requirement can be about two-thirds of that amount, depending on the speaker. The following table shows these disk requirement values:

	X-Bus 8kHz/6kHz	X-Bus 8kHz/6kHz
Pause Compression	No	Yes
Sampling Rate	8kHz / 6kHz	8kHz / 6kHz
Data Rate Bytes/Second (in thousands)	4 / 3	~2.7 / ~2
Bytes/Minute (in thousands)	240 / 180	~160 / ~120
Minutes/Megabyte	4.37 / 5.83	~6.55 / ~8.74

Memory Usage

When recording or playing back voice files, the application program must supply a work area for the Telephone that is 13,312 bytes or larger (in 1024-byte increments). The work area is divided into an 8192-byte data queue used by the hardware, and the balance (a minimum of 5120 bytes) is used for two file-system I/O buffers.

Additional memory improves performance. The disk performance is largely determined by disk seek time, which occurs once per I/O regardless of the number of bytes transferred. The larger the buffer sizes, the more bytes transferred per I/O and the fewer disk seeks needed. Performance improves with more memory because the system has more time in which to process the data. (Although the average time to process the data remains the same, a longer time interval allows a more uneven distribution of CPU and disk usage to occur.)

Structure of a Voice File

A voice file is divided into 512-byte sectors. The first 512 bytes of the voice file are used for the Voice File Header (Table 14-5), which contains information about how the voice message was recorded, the size of the file, when the recording was made, and the telephone line on which the recording was made. Many separate voice messages can be put into one voice file. See the section below, "Multiple Voice Messages in One File."

Within each 512-byte record after the Voice File Header, the first six bytes (0 through 5) are used for information about the record. Bytes 6 through 511 of each record are used for the voice information. (Table 14-6 shows the structure of a Voice File Record.) The first four bytes of each record contain the accumulated *sample number* of the first sample in that record. The sample number can be thought of as the raw count of bytes of voice information in the file. In the Voice File Header (Table 14-5), *qSampleStart* and *qSampleMax* give the starting and ending sample numbers for a particular voice message in the file. A *logical file address (lfa)*, in contrast, refers to an absolute byte position within a file. *lfaStart* and *lfaMax* specify the absolute byte position of the beginning and end of a particular message in the voice file. Because of pause compression, the difference between *lfaMax* and *lfaStart* can be smaller than the difference between *qSampleMax* and *qSampleStart* for a particular voice message.

The Voice File Header is the same for the Telephone Service, except for the version field. A value of 1 in the version field indicates Adaptive Differential Pulse Code Modulation (ADPCM), which is the only form of compression used on X-Bus workstations.

Adaptive Differential Pulse Code Modulation (ADPCM)

Adaptive Differential Pulse Code Modulation (ADPCM) is a general method of moderately compressing and expanding digitized voice information. X-Bus workstations use ADPCM for recording and playback.

Voice Control Structure

The Voice Control Structure (Table 14-7) contains information such as the file handle of the voice file, starting and ending logical file addresses, and starting and ending sample numbers. The TsVoiceRecordToFile and TsVoicePlaybackFromFile operations both require the caller to pass a pointer to this structure.

Typical Sequence for Record/Playback

The typical sequence for voice recording to or playing back from a file is as follows:

1. Open the voice file.
2. Allocate a 13,312-byte or larger work area in memory for use of the Telephone Service.
3. Set up the Voice Control Structure. When playing back, use information from the voice file's Voice File Header(s).
4. Call TsVoiceRecordToFile or TsVoicePlaybackFromFile.

Multiple Voice Messages in One File

Up to 65,536 messages can be contained in one voice file. A Voice File Header (Table 14-5) contains information for each message on recording rate, starting and ending lfa, and so on. Each Voice File Header can contain information for 15 voice messages. In the first Voice File Header, the message field should contain the count of messages. In all subsequent Voice File Header(s), if any, this field is ignored.

Voice Playback from Memory

You can play back a voice message from a disk file or from memory. To play back from memory, call TsVoicePlaybackFromFile with the following parameters:

- *pWorkArea* points to a memory area where the first 8192 bytes are reserved, and the following 0 to 56,320 bytes contain 0 to 110 voice file records (512 bytes each).
- *sWorkArea* is the size of the above memory area (8192 + size of voice information)
- The following fields of the Voice Control Structure (Table 14-7) should be set as follows:

fh	0FFFFh
lfaStart	0
lfaMax	Size of the voice information, or (sWorkArea - 8192)
qSampleStart	*qSampleStart* value in the first Voice File Record

See Example 14-4 for an example of voice playback from memory.

Telephony

The telephony interface allows an application program to control directly the functions of the Voice Processor hardware, including the two telephone lines, telephone unit, DTMF generator, DTMF decoder, CPTD, CODEC, and optional modem. This, in effect, allows an application program full access to the standard two-line telephone unit and the two telephone lines. The following paragraphs describe key concepts involved in writing programs that use the telephony interface.

Voice and Data Lines

Both telephone lines can be used to send either voice (analog information) or data (digital information), but not both at the same time. If your system has two telephone lines, it is conventional to use line 1 for voice and line 2 for data, but this is not a requirement. If you have only one telephone line, that line can be used for either voice or data, but not for both at the same time.

Telephone Unit vs. Telephone Line

When you remove a *standard* telephone handset from the cradle, the telephone line is said to be *offhook*. The term *offhook* refers to this active state of the telephone, when it is connected to the PBX or telephone company. When you replace the telephone handset, the switch hook on the telephone unit is depressed, and the telephone line is said to be *onhook*.

When a telephone unit is connected to the Voice Processor Module, there is a distinction between the *telephone handset* being offhook and the *telephone line* being offhook. The TsOnHook and TsOffHook operations place the telephone line onhook and offhook, even when the telephone handset remains in place. Similarly, the TsDoFunction operation instructs the Telephone Service to perform standard telephone unit functions, such as picking up lines, placing them on hold, and hanging up. In such cases, even though the telephone handset is not physically offhook, the telephone line can be offhook.

Hold

Placing the currently active telephone line on hold disconnects the telephone unit from the telephone line, but leaves the telephone line offhook. Both TsDoFunction and TsHold can be used to place the selected line on hold.

Dialing

Telephone numbers can be dialed directly from the telephone unit attached to the Voice Processor, from an application program such as the Operator or **Telephone Status** command, or through the programmatic call TsDial. Spaces, hyphens, and parentheses are ignored and can be omitted.

Some numbers need special characters for the Telephone Service to complete the call. For example, some PBXs require a pause after requesting an outside line and before dialing the outside number, as follows:

```
9~9412233
```

Table 14-1 includes a complete list of special characters used with the Telephone Service.

TsDial connects the DTMF tone generator to the specified telephone line and dials a telephone number. (See Example 14-2, "Dialing," later in this section.) TsReadTouchTone is used to read DTMF (touch-tone) characters from a telephone line or from the telephone unit.

Table 14-1. Dial Characters

Character	Function
0-9, A-Y, a-y, *, #	These characters are the dial or DTMF characters normally found on the touch pad or rotary switch. Q, q, Z, and z are not mapped.
!A, !B, !C, !D	These are the DTMF characters A through D, which do not normally appear on the touch pad.
%	Switch to pulse dialing.
&	Switch to tone dialing.
@	Flash for the default flash time.
!@*x*	Flash for *x* units of 100 ms (*x* is a byte).
=	Wait for a dial tone.
!=	Wait for any tone.
?	Indicates busy/fast busy condition on line.
|	Indicates ringing, busy, or answered. This character can be configured. (See "Call Progress Tone Detection".)
~	Pause for the default pause time.
!~*x*	Pause for *x* units of 100 ms.
$*tl*	Generate part of a dual-tone multifrequency (DTMF) tone *t* for time *l* (units of 10 ms). The primary use of this sequence is to generate a single tone (for use in answering machines, for example). See also the section below, "Generating DTMF Tones."
space, (,), -, /	These characters are ignored (*space* is ASCII 20h).
^	Flash for the default flash time.
+	Wait for a data carrier signal.

Generating DTMF Tones

The tone generated by the character sequence *$tl* described in Table 14-1 is derived from a combination of the two frequencies specified by the upper (column) and lower (row) 4-bit nibbles of *t*. The valid values of the nibble are 0Eh, 0Dh, 0Bh, and 07h. If both upper and lower nibbles are valid, the tone generated is one of the 16 DTMF tones as specified in the following table:

	0E0h	0D0h	0B0h	070h
0Eh	1	2	3	A
0Dh	4	5	6	B
0Bh	7	8	9	C
07h	*	0	#	D

For example, a value of 0E7h will generate the tone for *. If only one of the nibbles is valid (such as 07h or 0E0h), then a single tone is generated. If neither nibble is valid, no tone is generated.

Internationalized Call Progress Tone Detection

When the Telephone Service is initialized, it first looks for the standard call progress-tones configuration file *[Sys]<Sys>TmCptr.cfg*. If all of the required entries are present and valid, they are loaded into the Telephone Service. If this file is not present, or if it cannot be opened by the Telephone Service, or is incomplete, the default values for the United States are used.

The fixed call progress detection system is turned on by inserting a question mark (?) in the dial string (see Table 14-1). The internationalized system is turned on by the vertical bar (|). A brief example of how to use the internationalized system is shown below.

To use a nondefault call progress-tones configuration file, first create the file. Then call TsLoadCallProgressTones to load the new call progress tones into the Telephone Service.

The *TmCptr.cfg* file should contain the following entries:

```
:RingTone1High:
:RingTone1Low:
:RingTone2High:
:RingTone2Low:
:RingToneTolerance:
:nRingsReturnNoAnswer:
:BusyTone1High:
:BusyTone1Low:
BusyTone2High:
BusyTone2Low:
:BusyToneTolerance:
:CptrTimeout:
```

The values for the RingTone and BusyTone entries are integers, specified in milliseconds. The RingTones and BusyTones are defined by either one or two high/low cycles. For instance, in the United States, the ringing signal has a cycle of six seconds: two seconds on and four seconds off. Thus the entries are as follows:

```
:RingTone1High:2000
:RingTone1Low:4000
:RingTone2High:0
:RingTone2Low:0
```

In New Zealand, however, the ring tone has two cycles: .4 seconds on, .2 seconds off; .4 seconds on, 2 seconds off. Its entries are as follows:

```
:RingTone1High: 400
:RingTone1Low: 200
:RingTone2High: 400
:RingTone2Low: 2000
```

The *:RingToneTolerance:* and *:BusyToneTolerance:* entries are specified as percentages. These values indicate how much the duration of the cycle can vary from what was specified. Ten percent tolerance is usually adequate for most telephone networks.

The *:nRingsReturnNoAnswer:* field is used to return the TsDial request when the specified number of rings have been detected. Although this field can be set to any value, it should be at least 4 or 5 to allow the called party to reach the telephone.

The *:CptrTimeout:* field specifies the maximum number of seconds to perform nationalized call progress tone detection. Make sure that this field is greater than the cycle of the longest call progress tone multiplied by the value for *:nRingsReturnNoAnswer:*.

Default values for the United States are

```
:RingTone1High:2000
:RingTone1Low:4000
:RingTone2High:0
:RingTone2Low:0
:RingToneTolerance:10
:nRingsReturnNoAnswer:5
:BusyTone1High:500
:BusyTone1Low:500
:BusyTone2High:0
:BusyTone2Low:0
:BusyToneTolerance:10
:CptrTimeout:30
```

Examples of values used for other countries are listed below:

	Germany	France	Spain	Japan
RingTone1High	1000	1660	1500	1000
RingTone1Low	4000	3330	3000	2000
BusyTone1High	500	500	200	500
BusyTone1Low	500	500	200	500

Sample Program Using Internationalized Call Progress-Tone Detection System

In the following example bold statements indicate telephone service operations, which are explained in the *CTOS Procedural Interface Reference Manual.*

Example 14-1. Call Progress-Tone

```
/*
 * Assume:
   Filename is "[Sys]<Sys>TmCptr.cfg," or some other name
   such as "[Sys]<Sys>TmCptr.cfg=XYZ" where XYZ is the
   abbreviation for a country
 *
 * DialString is "~9~3957186|"
 */
#define lercTsTimeOut                                    11206
#define lercTsCptrFileNotFound                           11214
#define lercTsCptrFileIncomplete                         11215
#define lercTsCptrBusy                                   11216
#define lercTsCptrUnknownTone                            11217
#define   lercTsCptrAnswered                             11218
#define lercTsDialNoAnswer                               11278

/*
 * Load the customized call progress file.
 * This request will return either:
 *   ercOK
 *   lercTsCptrFileNotFound or
 *   lercTsCptrFileIncomplete
 */
erc = TsLoadCallProgressTones (iVpModule,
         Filename.pString, Filename.cbString);

/*
 * Take the telephone line off hook
 */
erc = TsDoFunction (iVpModule, iLine);
erc = TsDial (iVpModule, iLine, DialString.pString,
         DialString.cbString, cErrorTimeout, &wStatus));
```

Example 14-1. Call Progress-Tone (cont.)

```
/*
 * The ercs returned by TsDial include:
 * lercTsTimeOut
 *    This erc is returned when the CptrTimeout variable
 *    (in seconds) is reached.
 * lercTsCptrBusy
 *    The TsDial call returns immediately when the busy
 *    signal is detected.
 * lercTsCptrUnknownTone
 *    A tone of length not known to the system has been
 *    detected.  This erc may mean that additional tuning
 *    of the values loaded with the
 *    TsLoadCallProgressTones request is needed.  It may
 *    also mean that the called party has answered the
 *    phone.  The frequencies and cadence of a person
 *    saying "Hello" may be beyond the scope of the
 *    signal processing code.
 * lercTsCptrAnswered
 *    This erc is returned when the ringing signal has
 *    been detected, and then it has stopped.
 * lercTsDialNoAnswer
 *    This erc is returned when the line has rung the
 *    amount of times specified in the
 *    nRingsReturnNoAnswer field of the call progress
 *    tones configuration file.
 *
 * Based on the erc returned, the application either
 * plays the outgoing message or begins the next outbound
 * call.
 */
```

Data

The data interface allows application programs to use the optional modem in the Voice Processor Module available on X-Bus workstations. This modem allows the workstation and another computer or terminal with a compatible modem to transfer data over a telephone line. Either telephone line can be used to transmit data. Example 14-5 in the "Program Examples" section is an example of a data call.

Starting a Data Call

A data line is initiated with TsDataOpenLine. Data is transferred with either TsDataRead or TsDataWrite. A line is closed with TsDataClose. TsDataCheckpoint is used to checkpoint the data, allowing a program to guarantee transmission of all data. TsDataChangeParams and TsDataRetrieveParams are used to modify or examine control parameters such as parity or line control. TsDataUnacceptCall is used to retrieve a TsDataOpenLine request that is waiting to answer an incoming call.

The TsDataOpenLine service returns a handle that is used by subsequent data operations.

Use of the modem by a system service running under a single partition version of CTOS requires that the *fLL* flag in the Data Control Structure (Table 14-8) be set to TRUE. In all other cases, *fLL* should be FALSE.

Converting a Voice Call to a Data Call

An existing voice call can be converted to a data call. To do so, call TsDataOpenLine and set *openMode* in the Data Control Structure (Table 14-8) to 0. In this case, the specified telephone line must be offhook. The modem is connected to it, and the telephone unit and/or CODEC are disconnected if necessary. The *fOriginate* flag can be set to TRUE or FALSE, depending on which modem (remote or local) acts as originator. One modem must be the originator and one must be the answerer.

Accepting a Data Call

A data call is accepted by using the TsDataOpenLine service and setting openMode in the Data Control Structure to 1 and the *fOriginate* flag to FALSE. When the specified telephone line rings, the line is placed offhook and connected to the modem.

Originating a Data Call

A data call is originated by using the TsDataOpenLine service and setting *openMode* to 2 and the *fOriginate* flag to TRUE. The telephone line must be onhook. It is then placed offhook and a number is dialed. The number to be dialed can contain special characters, as described in Table 14-1. Unless the *fNoWaitForDialTone* flag in the Data Control Structure is set to TRUE (Table 14-8), the Telephone Service waits for a dial tone before dialing.

Reading and Writing Data

Most applications use the Voice Processor modem asynchronously to read and write data at the same time. Asynchronous processing is accomplished in one of two ways:

- By having two processes in the application that each use the procedural interface to TsDataRead and TsDataWrite
- By having one process that issues TsDataRead and TsDataWrite requests with the Request kernel primitive and then waits for responses to the TsDataRead and TsDataWrite requests with the Wait kernel primitive

Terminating a Data Call

A data call is terminated when a TsDataClose occurs, when the program using the modem terminates, or when a nonrecoverable error occurs. The following status codes indicate errors that are recoverable:

11205	Invalid handle (all)
11260	Duplicate request (TsDataCheckpoint)
11287	Data overrun (TsDataRead)
11288	Bad parity (TsDataRead)
11289	Data timeout (TsDataRead, TsDataWrite, or TsDataCheckpoint)
11290	End of block character (TsDataRead)

If a call is terminated due to a nonrecoverable error, subsequent operations using the same line handle will be returned with the terminating status code (until a TsDataCloseLine occurs). A TsDataCloseLine must be issued or the application program terminated, before the modem can be used again.

Telephone Status Debugging Tool

Use the **Telephone Status** command to verify the proper operation of Voice/Data Services or to debug programs that use the Voice Processor Module hardware. The **Telephone Status** command provides a simplified visual image of the hardware circuits that are connected when a telephone management function is performed.

This command is implemented with the run file *TmStatus.run*, command case TS.

Command Form

```
Telephone Status
  [Module number]
```

Parameter Field

[Module number]

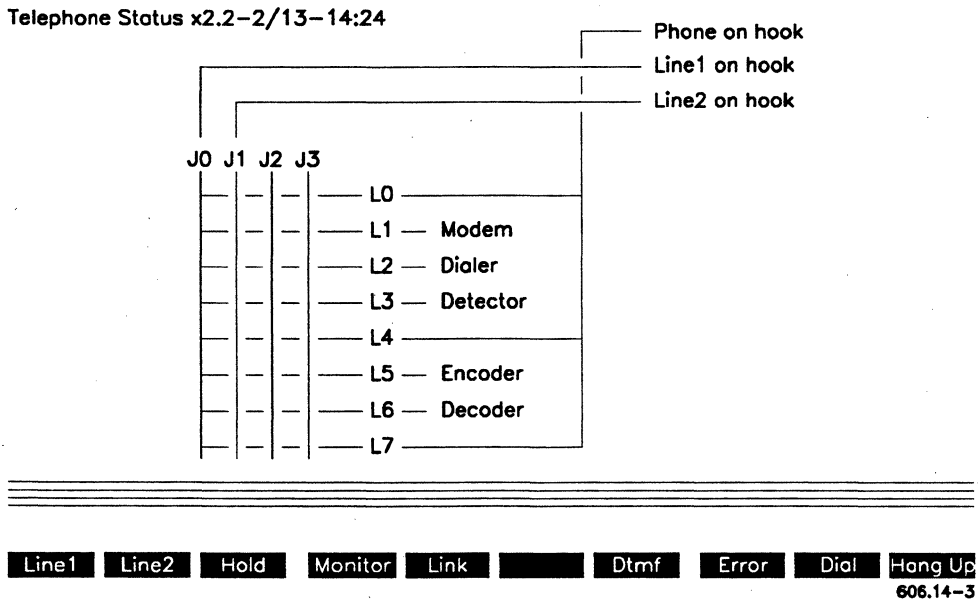
Default: 1

If your workstation has more than one Voice Processor Module, enter the appropriate module number. The Voice Processor Module closest to the CPU is 1. The second Voice Processor Module is 2, and so on.

Operation

After you fill in the command form and press **GO**, the screen appears as shown in Figure 14-3.

Figure 14-3. Telephone Status Command Screen



The upper righthand corner of the screen shows the three lines that represent the telephone unit connection and the two external telephone lines attached to the plug-in jacks on the back of the Voice Processor Module. (See the voice processor manual for more information about connections to the Voice Processor Module.)

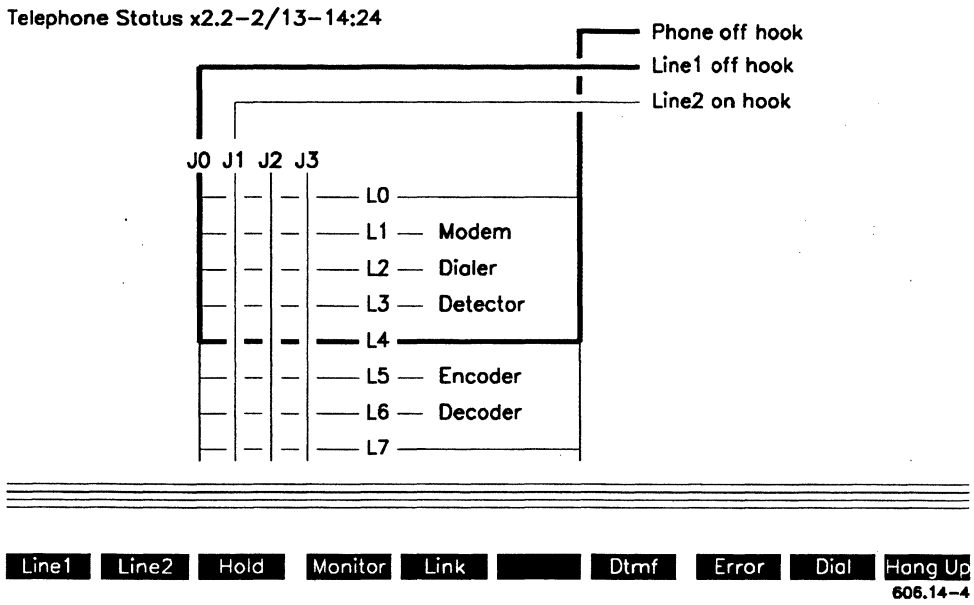
By using the soft function keys shown along the bottom of the screen, you can perform several telephone functions and watch a representation of the connections being made inside the Voice Processor Module. As circuits are completed and functions performed, lines on the screen light up to show a map of the internal circuitry of the module.

Lines J0 through J3

Lines J0 and J1 are the connecting lines between telephone lines 1 and 2 and other Voice Processor circuits, such as the connection for the telephone unit. They are lit when incoming calls pass through the internal hardware to make a connection with the telephone, as shown in Figure 14-4. Line J0 corresponds to telephone line 1, and Line J1 corresponds to telephone line 2.

Lines J2 and J3 are auxiliary lines that act as jumpers to facilitate internal hardware switching. Telephone calls to the modem, dialer, or CODEC, for example, are connected to lines J2 and J3. These lines also light up to show you how the internal switching takes place.

Figure 14-4. Telephone Status Screen with Telephone Offhook



Lines L0 through L7

The functions of lines L0 through L7 are summarized below.

Line Number	Function	Description
L0 (Telephone Unit)		Is a straight-through connection to the telephone unit, without any modulation of the signal.
L1	Modem	Is the connection between the telephone lines and the optional modem.
L2	Dialer	Is the connection from the DTMF touch-tone dialer to the telephone lines and/or telephone unit.
L3	Detector	Is the connection to the detector, which senses activity on the incoming and outgoing lines, such as busy signals and dial tones.
L4 (Telephone Unit)		Is a connection that amplifies voice signals to and from the telephone unit for improved clarity.
L5	Encoder	Is the connection to the encoder, which converts analog (voice) signals to digital (binary) signals so that the workstation can process voice messages as file data.
L6	Decoder	Is the connection to the decoder, which converts digital (binary) signals to analog (voice) signals.
L7 (Telephone Unit)		Is a connection to the telephone unit that attenuates the high-volume DTMF dialing signals generated inside the module.

Status Monitor Function Keys

The functions performed by the Telephone Status function keys are summarized below:

Key	Label	Explanation
F1	Line 1	Makes the connection between the telephone unit and telephone line 1.
F2	Line 2	Makes the connection between the telephone unit and telephone line 2.
F3	Hold	Places the currently active telephone line on hold. Pressing **F1** or **F2** takes the telephone line off hold (if it is currently on hold).
F4	Monitor	Allows the screening of incoming calls (designed for use with an application software's answering machine function).
F5	Link	Allows conference calling by connecting the telephone unit, telephone line 1, and telephone line 2.
F7	DTMF	Toggles on/off the reading of touch-tone characters. When the DTMF characters are detected, they are displayed on the screen.
F8	Error	Displays the request block of the most recent request that the Telephone Service returned with an error.
F9	Dial	Allows you to type a number on the keyboard and then dial it by pressing **F9**. (This can replace, but will not obstruct, dialing from the telephone unit.)
F10	Hang up	Disconnects the telephone line from the telephone unit and hangs it up.

Program Examples

This section contains program examples that show how to use voice/data services on X-Bus workstations. These examples were written for the Metaware High C Compiler.

Example 14-2: Dialing

The following program uses the telephony features of the Telephone Service and Voice Processor Module to dial a telephone number. In this program, the TsGetStatus call returns the telephone status (see Table 14-4, for descriptions of this data structure, which includes *LineState* and *tUnitState*.)

The example uses TsDial twice: first to wait for a dial tone and second, to dial the number. A comment shows how to call TsDial multiple times to dial separate parts of a dialing sequence.

Note the section that begins with the pragma statement. This section is required because the C language uses a different parameter-passing convention than CTOS uses. See your compiler manual for information about your compiler's calling conventions. For more information on the calling conventions used by CTOS, see "Stack Format and Calling Conventions" in *CTOS / Open Programming Practices and Standards*.

Example 14-2. Dial.c

```
/*   Program title:  Dial.c
 *   Compiler:  Metaware High C Compiler
 *
 * Dial
 *   [Number]
 */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
```

Example 14-2. Dial.c (cont.)

```
#define Syslit
#define sdType
#include <CTOSTypes.h>

#define CheckErc
#define ErrorExit
#define RgParam
#include <CTOSLib.h>
/*
 * This is one of the fields of TSStatusType
 */
struct LineState {
   char status;
   char fDialing;
   char dialState;
   char fRinging;
   char iRing;
   char fRingThrough;
   char fOffHook;
   char fHold;
   char fCodec;
   char fDtmfRec;
   int handle;
   char reserved;
   char fModem;
   char reserved2[2];
};

struct TSStatusType {
   int iEvent;
   char defaultLine;
   char fNeedCodecConnection;
   char fCodecInUse;
   char aTmcb[3];
   int reserved;
   int baudrate;
   char originateMode;
   char parityMode;
   char lineControlMode;
   char reserved2;
```

Example 14-2. Dial.c (cont.)

```
    /*
     * the following are part of tUnitState
     */
    char fOffHook;
    char rgfTLine[2];
    char rgfRingThrough[2];
    char fCodec;
    char fDtmfRec;
    char fMonitor;
    int handle;
    char hookThroughMode;
    char reserved3[5];
    /*
     * end of tUnitState
     */
    struct LineState rgLineState[2];
    /*
     * codec state
     */
    long lfaCurrent;
    long qSampleCurrent;
    char reserved4[8];
    /*
     * end of codec state
     */
};

pragma Calling_convention(CTOS_CALLING_CONVENTIONS);

int TsDial(int liVpModule, int line, void *pbString,
    int cbString, int cErrorTimeOut,
    void *pcbStringRet);
int TsOffHook(int liVpModule, void *pThRet,
    int line);
int TsGetStatus(int liVpModule, void *pStatusRet,
    int sStatusRetMax, int fNoWait);
```

Example 14-2. Dial.c (cont.)

```
pragma Calling_convention();

#define liVpModule 1
#define liLine 0void main()

{
   int th, cbDialRet;
   char fPrompted = FALSE;
   sdType sdParam;
   struct TSStatusType TSStatus;
   /*
    * Get phone number
    */
   CheckErc(RgParam(1, 0, &sdParam));
   /*
    *  See if telephone unit is connected to liLine.
    *  If not, then prompt the user and wait until
    *  it is.
    */
   do {
      CheckErc(TsGetStatus(liVpModule, &TSStatus,
      sizeof(struct TSStatusType), FALSE));
      if ((!fPrompted) && (TSStatus.fOffHook
         == FALSE)) {
         printf("Please pick up the handset...\n");
         fPrompted = TRUE;
      }
   } while (TSStatus.fOffHook == FALSE);
   /*
    * Take phone off hook
    */
   CheckErc(TsOffHook(1, &th, 0));
   /*
    *  Wait for a dial tone
    */
   CheckErc(TsDial(liVpModule, liLine, "=", 1, 100,
      &cbDialRet));
```

Example 14-2. Dial.c (cont.)

```
    /*
     * Dial number, the following code fragment shows that
     * multiple TsDial calls can be made to dial the
     * number, one character at a time.  However, we will
     * do it all at once now that TsDial has been used to
     * wait for a dial tone.
     *
     * for (i = 0; i < sdParam.cb; i++) {
     *     CheckErc(TsDial(liVpModule, liLine,
                    sdParam.pb++,
     *     1, 100, &cbDialRet));
     * }
     */
    CheckErc(TsDial(liVpModule, liLine, sdParam.pb,
              sdParam.cb,
        100, &cbDialRet));
}
```

Example 14-3: Voice Response System

This example answers the telephone, plays an initial message, and then plays other messages in response to DTMF input. If, at any time during the playing of a message, a DTMF character is received, the current message is stopped, and the message corresponding to the new DTMF character begins. In a voice response application, **ACTION+FINISH** should be disabled and the typing of the **FINISH** key should be an asynchronous event that is recognized.

Setting Up Request Blocks

This program shows use of the request procedure interface, as well as use of direct requests to the Telephone Service. Request blocks are set up for TsReadTouchTone and TsVoicePlaybackFromFile. These two requests are made at the same time. As the initial voice file is playing, the Telephone Service is waiting at an exchange. (The default exchange cannot be used when using the Request and Wait kernel primitives.) One of two events occurs. Either TsVoicePlaybackFromFile returns first and the *fPlayBackFinished* flag is set to TRUE, or TsReadTouchTone returns first. If TsReadTouchTone returns first, either a DTMF character was received or a timeout occurred without a character having been read. If a DTMF character was received, the message currently being played (if any) is stopped, and the message for the new DTMF character is played. In addition, any functions associated with that character are carried out, for example, hanging up after the # message is played.

fNoWait Flag

In cases where you want TsGetStatus to return the status immediately, set the *fNoWait* flag to TRUE. If you want status returned only when a change in status occurs, set this flag to FALSE. This sample program illustrates both cases. In the WaitForCall function, *fNoWait* is set to FALSE, which causes TsGetStatus to wait until there is a change in status. In the StartResponse function, *fNoWait* is set to TRUE, because we need to know immediately whether the CODEC is already in use. Since the CODEC cannot be shared, you must be sure that it is free before calling TsDoFunction to lock the CODEC.

Use of TsVoiceConnect

In the ConnectVoice function, TsVoiceConnect specifies the connection of the telephone line to the CODEC. Because the *fAutoStart* flag in the Voice Control Structure is FALSE, TsVoiceConnect must be called before each call to TsVoicePlaybackFromFile, as shown below.

Possible Modifications to This Program

For purposes of the example, this program reads and responds to one DTMF character at a time. A more sophisticated system could be built around this one. For example, if the user needs to enter a string of numbers, such as a telephone extension, the system might read the first character into a buffer and then immediately reissue additional requests to TsReadTouchTone until the complete string is received. Reading the characters one-by-one enables the system to make an immediate response to the first character, for example, by turning off the recorded message as soon as the user enters the first number of the extension.

Example 14-3. Response.c

```
/*
 * Program title:  Response.c
 * Compiler:  Metaware High C Compiler
 *  *
 * Response
 *   [iVpModule      ]
 *   [Line to answer]
 *   [Answer on ring]
 *   [First Message ]
 *   [DTMF 1 Message]
 *   [DTMF 2 Message]
 *   [DTMF # Message]
 *   [DTMF Error Msg]
 *
 *  Explanation of command form options:
 *
 *     iVpModule:       Voice Processor Module number
 *                      (default = 1)
 *     Line to answer: Either Line 1 or Line 2
 *                      (default = 1)
```

Example 14-3. Response.c (cont.)

```
 *      Answer on ring: Answer the phone after this many
 *                      rings.  (default = 1)
 *      First Message:  The initial message.
 *      DTMF ? Message: Message played when the
 *                      corresponding DTMF character
 *                      is read.
 *      DTMF # Message: Stops the playing of any message,
 *                      plays the hangup message and
 *                      hangs up the phone.
 *      DTMF Error Msg: The message that is played if a
 *                      DTMF digit is read for which
 *                      there is no file to playback or if
 *                      the DTMF read operation has timed
 *                      out.
 */
#include <string.h>
#include <stdlib.h>
#define RqHeaderType
#define Syslit
#define sdType
#include <CTOSTypes.h>
/*
 * Voice Control Structure
 */
struct VCStype {
   int fh;
   long lfaStart;
   long lfaMax;
   long qSampleStart;
   long qSampleMax;
   int cPauseMax;
   int cSampleOn;
   int cSampleOff;
   char f6KHz;
   char fAutoStart;
   char fNoPause;
   char fStopOnDialTone;
   char fAltConnection;
   int nSectorStatusUpdate;
   int sPauseGap;
   char fRawData;
};
```

Example 14-3. Response.c (cont.)

```
/*
 * Voice File Entry
 */
struct VDFHEntrytype {
   char f6KHz;
   long lfaStart;
   long lfaMax;
   long qSampleStart;
   long qSampleMax;
   char bReserved;
   long dateTime;
   char line;
   char fNoPause;
   char fAltConnection;
   char rgReserved[7];
};
/*
 * Voice File Header
 */
struct VDFHtype {
   int signature;
   int version;
   int wReserved;
   struct VDFHEntrytype message[15];
   char rgReserved[26];
};
/*
 * LineState is used in TSStatusType below
 */
struct LineState {
   char status;
   char fDialing;
   char dialState;
   char fRinging;
   char iRing;
   char fRingThrough;
   char fOffHook;
   char fHold;
```

Example 14-3. Response.c (cont.)

```
    char fCodec;
    char fDtmfRec;
    int handle;
    char reserved;
    char fModem;
    char reserved2[2];
};
/*
 * Telephone Service Status
 */
struct TSStatusType {
    int iEvent;
    char defaultLine;
    char fNeedCodecConnection;
    char fCodecInUse;
    char aTmcb[3];
    int reserved;
    int baudrate;
    char originateMode;
    char parityMode;
    char lineControlMode;
    char reserved2;
    char fOffHook;  /* beginning of tUnitState */
    char rgfTLine[2];
    char rgfRingThrough[2];
    char fCodec;
    char fDtmfRec;
    char fMonitor;
    int handle;
    char hookThroughMode;
    char reserved3[5]; /* end of tUnitState */
    struct LineState rgLineState[2];
    long lfaCurrent; /* beginning of codec state */
    long qSampleCurrent;
    char reserved4[8]; /* end of codec state */
};
```

Example 14-3. Response.c (cont.)

```
/*
 * Voice Processor Requests
 */
pragma Calling_convention(CTOS_CALLING_CONVENTIONS);
int TsDoFunction(int iVpModule, int function);
int TsGetStatus(int iVpModule, void *pStatusRet,
   int  sStatusRetMax, int fNoWait);
int sOffHook(int iVpModule, void *pThRet, int line);
int TsOnHook(int iVpModule, int line);
int TsVoiceConnect(int iVpModule, char fVoiceUnit,
   char fLine0, char fLine1);
int TsVoicePlaybackFromFile(int iVpModule,
   void *pWorkArea, int sWorkArea, void *pVoiceControl,
   int sVoiceControl, void *pLfaLast,
   void *pqSampleLast, void *pStatusRet);
int TsVoiceStop(int iVpModule);
pragma Calling_convention();
/*
 * TsReadTouchTone request code and request block
 */
#define rcTsReadTouchTone 0x8030
struct rqTsReadTouchToneType {
   char sCntInfo;
   char RtCode;
   char nReqPbCb;
   char nRespPbCb;
   int userNum;
   int exchResp;
   int ercRet;
   int rqCode;
   int iVpModule;
   int device;
   int bStopDigit;
   int cTimeOut;
   char *pbDigits;
   int cbDigitsMax;
   char *pcbDigitsRet;
   int scbDigitsRet;
};
```

Example 14-2. Response.c (cont.)

```
/*
 * TsVoicePlaybackFromFile request code and request block
 */
#define rcTsVoicePlaybackFromFile 0x8021
struct rqTsVoicePlaybackFromFileType {
   char sCntInfo;
   char RtCode;
   char nReqPbCb;
   char nRespPbCb;
   int userNum;
   int exchResp;
   int ercRet;
   int rqCode;
   int iVpModule;
   char *pWorkArea;
   int sWorkArea;
   char *pVoiceControl;
   int sVoiceControl;
   long *plfaLast;
   int slfaLast;"
   long *pqSampleLast;
   int sqSampleLast;
   int *pStatusRet;
   int sStatusRet;
};

/*
 * CTOS Library calls
 */
#define AllocExch
#define AllocMemorySL
#define CheckErc
#define CloseFile
#define Delay
#define ErrorExit
#define OpenFile
#define Read
#define Request
#define RgParam
#define Wait
#include <CTOSLib.h>
```

Example 14-3. Response.c (cont.)

```
/*
 * constants
 */
#define lsbWorkArea 32768
#define lDigits 1
#define lParameterFound 0
#define lParameterNotFound 2450
#define lercDTMFTimeout 11206
#define lercRequestIncomplete 282
#define lLine1 1
#define lLine2 2
#define lercInvalidParameters 11203
/*
 * global variables
 */
struct TSStatusType TSStatus;  /* Used by TsGetStatus */
struct VDFHtype VDFH;  /* Voice file header structure */
struct VCStype VCS;  /* for TsVoicePlaybackFromFile */
struct rqTsVoicePlaybackFromFileType
   rqTsVoicePlaybackFromFile;
struct rqTsReadTouchToneType rqTsReadTouchTone;
RqHeaderType *pRequestRet;
char fAnswer = FALSE, rgbParam[64], *pWorkArea,
   rgbDigits[lDigits], fPlaybackFinished = FALSE,
   fDTMFReceived = FALSE, fDTMFTimeout = FALSE,
   fContinue = TRUE;
int iVpModule, iLine, nAnswerRings, erc, fhInitial,
   fhDTMF1, fhDTMF2, fhDTMFPound, fhError,
   TSExchange, StatusRet, th, cbRead, cbDigitsRet;
long lfaEnd, SampleEnd;

/*******************************************************/
int main()

{
   ErrorCleanup(GetParams());
   /*
    * Allocate exchange for Telephone Service responses.
    */
   ErrorCleanup(AllocExch(&TSExchange));
```

Example 14-3. Response.c (cont.)

```
    /*
     * Buffer for Telephone Service voice playback.
     */
    ErrorCleanup(AllocMemorySL(lsbWorkArea, &pWorkArea));
    while (TRUE) {
       ErrorCleanup(WaitForCall()); /* Wait for call */
       /*
        * Check to make sure that the telephone unit is
        * not connected to the line we want to answer.
        * If so then don't answer the phone.
        */
       ErrorCleanup(Delay(2));
       ErrorCleanup(TsGetStatus(iVpModule, &TSStatus,
          sizeof(struct TSStatusType), TRUE));
       if(TSStatus.rgfTLine[iLine - 1] == FALSE) {
          ErrorCleanup(StartResponse());
          ErrorCleanup(CleanupLine());
       }
    }
    return(ercOK);
}
/*******************************************************/
int WaitForCall()
{
    while (!fAnswer) {
       /*
        * If TsGetStatus is called with the fNoWait FALSE,
        * call returns when telephone event occur.
        */
       erc = TsGetStatus(iVpModule, &TSStatus,
          sizeof(struct TSStatusType), FALSE);
       if (erc != ercOK) return erc;
       /*
        * Is the event a ringing telephone line?
        */
       if (TSStatus.rgLineState[iLine-1].fRinging
          != FALSE) {
          if (TSStatus.rgLineState[iLine - 1].iRing
             >= nAnswerRings) fAnswer = TRUE; /* answer */
       }
    }
```

Example 14-3. Response.c (cont.)

```
    fAnswer = FALSE; /* Reset flag for next call */
    return erc;
}

/*******************************************************/
int GetParams()
{
    sdType sdParam; /* for RgParam calls */
    erc = RgParam(1, 0, &sdParam);
    strncpy(rgbParam, sdParam.pb, sdParam.cb);
    switch(erc) {
        case lParameterFound:
            rgbParam[sdParam.cb] = '\0';
            iVpModule = atoi(rgbParam);
            if ((iVpModule < 1) || (iVpModule > 5))
                iVpModule = 1;
            break;
        case lParameterNotFound:
            iVpModule = 1;
            break;
        default:
            return erc;
    }
    erc = RgParam(2, 0, &sdParam);
    strncpy(rgbParam, sdParam.pb, sdParam.cb);
    switch(erc) {
        case lParameterFound:
            rgbParam[sdParam.cb] = '\0';
            iLine = atoi(rgbParam);
            if ((iLine < 1) || (iLine > 2)) iLine = 1;
            break;
        case lParameterNotFound:
            iLine = 1;
            break;
        default:
            return erc;
    }
    erc = RgParam(3, 0, &sdParam);
    strncpy(rgbParam, sdParam.pb, sdParam.cb);
```

Example 14-3. Response.c (cont.)

```
switch(erc) {
   case lParameterFound:
      rgbParam[sdParam.cb] = '\0';
      nAnswerRings = atoi(rgbParam);
      if ((nAnswerRings < 1) || (nAnswerRings > 10))
         nAnswerRings = 2;
      break;
   case lParameterNotFound:
      nAnswerRings = 1;
      break;
   default:
      return erc;
}
erc = RgParam(4, 0, &sdParam);
switch(erc) {
   case lParameterFound:
      erc = OpenFile(&fhInitial, sdParam.pb,
         sdParam.cb, "", 0, modeRead);
      if (erc != ercOK) return erc;
      break;
   case lParameterNotFound:
      strncpy(rgbParam, "Initial.Voice", 13);
      sdParam.cb = 13;
      erc = OpenFile(&fhInitial, rgbParam, sdParam.cb,
         "", 0, modeRead);
      if (erc != ercOK) return erc;
      break;
   default:
      return erc;
}
erc = RgParam(5, 0, &sdParam);
switch(erc) {
   case lParameterFound:
      erc = OpenFile(&fhDTMF1, sdParam.pb, sdParam.cb,
         "", 0, modeRead);
      if (erc != ercOK) return erc;
      break;
   case lParameterNotFound:
      strncpy(rgbParam, "DTMF1.Voice", 11);
      sdParam.cb = 11;
```

Example 14-3. Response.c (cont.)

```
        erc = OpenFile(&fhDTMF1, rgbParam, sdParam.cb,
            "", 0, modeRead);
        if (erc != ercOK) return erc;
        break;
    default:
        return erc;
}
erc = RgParam(6, 0, &sdParam);
switch(erc) {
    case lParameterFound:
        erc = OpenFile(&fhDTMF2, sdParam.pb, sdParam.cb,
            "", 0, modeRead);
        if (erc != ercOK) return erc;
        break;
    case lParameterNotFound:
        strncpy(rgbParam, "DTMF2.Voice", 11);
        sdParam.cb = 11;
        erc = OpenFile(&fhDTMF2, rgbParam, sdParam.cb,
            "", 0, modeRead);
        if (erc != ercOK) return erc;
        break;
    default:
        return erc;
}
erc = RgParam(7, 0, &sdParam);
switch(erc) {
    case lParameterFound:
        erc = OpenFile(&fhDTMFPound, sdParam.pb,
            sdParam.cb, "", 0, modeRead);
        if (erc != ercOK) return erc;
        break;
    case lParameterNotFound:
        strncpy(rgbParam, "DTMFPound.Voice", 15);
        sdParam.cb = 15;
        erc = OpenFile(&fhDTMFPound, rgbParam,
            sdParam.cb, "", 0, modeRead);
        if (erc != ercOK) return erc;
        break;
    default:
        return erc;
}
```

Example 14-3. Response.c (cont.)

```
    erc = RgParam(8, 0, &sdParam);
    switch(erc) {
       case lParameterFound:
          erc = OpenFile(&fhError, sdParam.pb, sdParam.cb,
             "", 0, modeRead);
          if (erc != ercOK) return erc;
          break;
       case lParameterNotFound:
          strncpy(rgbParam, "Error.Voice", 11);
          sdParam.cb = 11;
          erc = OpenFile(&fhError, rgbParam, sdParam.cb,
             "", 0, modeRead);
          if (erc != ercOK) return erc;
          break;
       default:
          return erc;
    }
    return(ercOK);
}
/*******************************************************/
int StartResponse()
{
    /*
     * Don't answer phone if codec is in use.  TsGetStatus
       called with fNoWait TRUE returns immediately.
     */
    erc = TsGetStatus(iVpModule, &TSStatus,
       sizeof(struct TSStatusType), TRUE);
    if ((erc != ercOK) || (TSStatus.fCodecInUse == TRUE))
       return erc;
    erc = TsDoFunction(iVpModule, 12); /* Lock codec */
    if (erc != ercOK) return erc;
    /*
     * Take phone off hook.
     */
    erc = TsOffHook(iVpModule, &th, (iLine - 1));
    if (erc != ercOK) return erc;
    erc = ConnectVoice();
    if (erc != ercOK) return erc;
    erc = PlayMessage(fhInitial); /* play first msg */
    if (erc != ercOK) return erc;
```

Example 14-3. Response.c (cont.)

```
/*
 * Now wait for returned requests.  Two cases:
 *
 *   a.    TsReadTouchTone is returned first.  Stop the
 *         currently playing message and do whatever
 *         the DTMF digit instructs.
 *   b.    TsVoicePlaybackFromFile is returned first.
 *         Just turn on a flag that request has been
 *         returned and rewait for DTMF respond.
 */
while (fContinue) {
   erc = Wait(TSExchange, &pRequestRet);
   if (erc != ercOK) return erc;
   if (pRequestRet->rqCode
      == rcTsVoicePlaybackFromFile)
      VoicePlaybackRespond(); /* Playback respond */
   else if (pRequestRet->rqCode ==
      rcTsReadTouchTone)
      ReadTouchToneRespond(); /* ReadTT respond */
   else return(17); /* "Mismatched respond" */
   /*
    * A DTMF character has been received or the voice
    * file finished playing.  Wait for DTMF chars and
    * play messages until timeout or # key is pressed.
    */
   if ((fPlaybackFinished == FALSE) &&
      (fDTMFReceived == TRUE)) {
      erc = TsVoiceStop(iVpModule);
      if (erc != ercOK) return erc;
      erc = Wait(TSExchange, &pRequestRet);
      if (erc != ercOK) return erc;
      /*
       * should be TsVoicePlaybackFromFile request
       */
      if (pRequestRet->rqCode !=
         rcTsVoicePlaybackFromFile) return(17);
```

Example 14-3. Response.c (cont.)

```
switch(rgbDigits[0]) {
   case '1'..'2':
      erc = ConnectVoice();
      if (erc != ercOK) return erc;
      erc = PlayMessage(MapDTMFtoFH
         (rgbDigits[0]));
      if (erc != ercOK) return erc;
      break;
   case '#':
      erc = SetupVoiceFileVCSForPlayback
         (fhDTMFPound);
      if (erc != ercOK) return erc;
      erc = ConnectVoice();
      if (erc != ercOK) return erc;
      erc = TsVoicePlaybackFromFile(iVpModule,
         pWorkArea, lsbWorkArea, (char *)(&VCS),
         sizeof(struct VCStype), &lfaEnd,
         &SampleEnd, &StatusRet);
      if (erc != ercOK) return erc;
      fContinue = FALSE;
      break;
   /*
    * default if DTMF is not 1, 2, or #.
    */
   default:
      erc = SetupVoiceFileVCSForPlayback
            (fhError);
      if (erc != ercOK) return erc;
      erc = ConnectVoice();
      if (erc != ercOK) return erc;
      erc = TsVoicePlaybackFromFile(iVpModule,
          pWorkArea, lsbWorkArea,(char *)(&VCS),
          sizeof(struct VCStype), &lfaEnd,
          &SampleEnd, &StatusRet);
      if (erc != ercOK) return erc;
      fContinue = FALSE;
}
```

Example 14-3. Response.c (cont.)

```
} else if (fDTMFTimeout) {
   /*
    * Play error message and hang up.  No DTMF.
    */
   erc = TsVoiceStop(iVpModule);
   if (erc != ercOK) return erc;
   erc = Wait(TSExchange, &pRequestRet);
   if (erc != ercOK) return erc;
   /*
    * TsVoicePlaybackFromFile request
    */
   if (pRequestRet->rqCode !=
      rcTsVoicePlaybackFromFile) return(17);
      erc = SetupVoiceFileVCSForPlayback(fhError);
      if (erc != ercOK) return erc;
      erc = ConnectVoice();
      if (erc != ercOK) return erc;
      erc = TsVoicePlaybackFromFile(iVpModule,
         pWorkArea, lsbWorkArea, (char *)(&VCS),
         sizeof(struct VCStype), &lfaEnd,
         &SampleEnd, &StatusRet);
      if (erc != ercOK) return erc;
   fContinue = FALSE;
} else if ((fPlaybackFinished == TRUE) &&
   (fDTMFReceived == FALSE)) {
   /*
    * wait for a character or timeout.
    */
   erc = Wait(TSExchange, &pRequestRet);
   if (erc != ercOK) return erc;
   /* if Timeout set flag. "while" handles it on
    * next pass;  otherwise call PlayDTMFMessage()
    */
   if (pRequestRet->rqCode != rcTsReadTouchTone)
      return(17);
   switch(rqTsReadTouchTone.ercRet) {
      case ercOK:
         break;
      case lercDTMFTimeout:
         fDTMFTimeout = TRUE;
         break;
```

Example 14-3. Response.c (cont.)

```
   default:
      return(rqTsReadTouchTone.ercRet);
}
if (fDTMFTimeout) {
   erc = SetupVoiceFileVCSForPlayback(fhError);
   if (erc != ercOK) return erc;
   erc = ConnectVoice();
   if (erc != ercOK) return erc;
   erc = TsVoicePlaybackFromFile(iVpModule,
      pWorkArea, lsbWorkArea, (char *)(&VCS),
      sizeof(struct VCStype), &lfaEnd,
      &SampleEnd, &StatusRet);
   if (erc  != ercOK) return erc;
   fContinue = FALSE;
} else {
   switch(rgbDigits[0]) {
      case '1'..'2':
         erc = ConnectVoice();
         if (erc != ercOK) return erc;
         erc = PlayMessage(MapDTMFtoFH
               (rgbDigits[0]));
         if (erc != ercOK) return erc;
         break;
      case '#':
         erc = SetupVoiceFileVCSForPlayback
            (fhDTMFPound);
         if (erc != ercOK) return erc;
         erc = ConnectVoice();
         if (erc != ercOK) return erc;
         erc = TsVoicePlaybackFromFile
            (iVpModule, pWorkArea,
            lsbWorkArea, (char *)(&VCS),
            sizeof(struct VCStype), &lfaEnd,
            &SampleEnd, &StatusRet);
         if (erc != ercOK) return erc;
         fContinue = FALSE;
         break;
      /*
       * default if DTMF is not 1, 2 or #.
       */
```

Example 14-3. Response.c (cont.)

```
                default:
                    erc = SetupVoiceFileVCSForPlayback
                          (fhError);
                    if (erc != ercOK) return erc;
                    erc = ConnectVoice();
                    if (erc != ercOK) return erc;
                    erc = TsVoicePlaybackFromFile
                       (iVpModule, pWorkArea, lsbWorkArea,
                       (char *)(&VCS), sizeof(struct
                       VCStype), &lfaEnd,
                       &SampleEnd, &StatusRet);
                    if (erc != ercOK) return erc;
                    fContinue = FALSE;
            }
        }
    }
    }
    fContinue = TRUE;
    fDTMFTimeout = FALSE;
    return(ercOK);
}
/*******************************************************/
int SetupVoiceFileVCSForPlayback(fh)
int fh;
{
    int erc;
    erc = Read(fh, &VDFH, sizeof(struct VDFHtype), 0,
      &cbRead);
    if (erc != ercOK) return erc;
    if (cbRead != 512) return(lercRequestIncomplete);
    VCS.fh = fh;
    VCS.lfaStart = VDFH.message[0].lfaStart;
    VCS.lfaMax = VDFH.message[0].lfaMax;
    VCS.qSampleStart = VDFH.message[0].qSampleStart;
    VCS.qSampleMax = VDFH.message[0].qSampleMax;
    VCS.cPauseMax = 0;
    VCS.cSampleOn = 0;
    VCS.cSampleOff = 0;
    VCS.f6KHz = VDFH.message[0].f6KHz;
    VCS.fAutoStart = FALSE;
    VCS.fNoPause = FALSE;
```

Example 14-3. Response.c (cont.)

```
    VCS.fStopOnDialTone = FALSE;
    VCS.fAltConnection = FALSE;
    VCS.nSectorStatusUpdate = FALSE;
    VCS.sPauseGap = 0;
    VCS.fRawData = FALSE;
    return(ercOK);
}

/*******************************************************/
int SetupVoiceFileRqForPlayback()
{
    rqTsVoicePlaybackFromFile.sCntInfo = 2;
    rqTsVoicePlaybackFromFile.RtCode = 0;
    rqTsVoicePlaybackFromFile.nReqPbCb = 2;
    rqTsVoicePlaybackFromFile.nRespPbCb = 3;
    rqTsVoicePlaybackFromFile.userNum = 0;
    rqTsVoicePlaybackFromFile.ercRet = 0;
    rqTsVoicePlaybackFromFile.exchResp = TSExchange;
    rqTsVoicePlaybackFromFile.rqCode =
       rcTsVoicePlaybackFromFile;
    rqTsVoicePlaybackFromFile.iVpModule = iVpModule;
    rqTsVoicePlaybackFromFile.pWorkArea = pWorkArea;
    rqTsVoicePlaybackFromFile.sWorkArea = lsbWorkArea;
    rqTsVoicePlaybackFromFile.pVoiceControl =
       (char *)(&VCS);
    rqTsVoicePlaybackFromFile.sVoiceControl =
       sizeof(struct VCStype);>
    rqTsVoicePlaybackFromFile.plfaLast = &lfaEnd;
    rqTsVoicePlaybackFromFile.slfaLast = 4;
    rqTsVoicePlaybackFromFile.pqSampleLast = &SampleEnd;
    rqTsVoicePlaybackFromFile.sqSampleLast = 4;
    rqTsVoicePlaybackFromFile.pStatusRet = &StatusRet;
    rqTsVoicePlaybackFromFile.sStatusRet = 2;
    return(ercOK);
}
/*******************************************************/
int SetupReadTouchToneRq()
{
    rqTsReadTouchTone.sCntInfo = 8;
    rqTsReadTouchTone.RtCode = 0;
    rqTsReadTouchTone.nReqPbCb = 0;
```

Example 14-3. Response.c (cont.)

```
    rqTsReadTouchTone.nRespPbCb = 2;
    rqTsReadTouchTone.userNum = 0;
    rqTsReadTouchTone.ercRet = 0;
    rqTsReadTouchTone.exchResp = TSExchange;
    rqTsReadTouchTone.rqCode = rcTsReadTouchTone;
    rqTsReadTouchTone.iVpModule = iVpModule;
    rqTsReadTouchTone.device = iLine - 1;
    rqTsReadTouchTone.bStopDigit = 0; /* illegal char */
    rqTsReadTouchTone.cTimeOut =180;
    rqTsReadTouchTone.pbDigits = &rgbDigits[0];
    rqTsReadTouchTone.cbDigitsMax = sizeof(rgbDigits);
    rqTsReadTouchTone.pcbDigitsRet = (char *)&cbDigitsRet;
    rqTsReadTouchTone.scbDigitsRet = 2;
    return(ercOK);
}
/*******************************************************/
int VoicePlaybackRespond()
{
    /*
     * Check request block for errors.
     */
    if (rqTsVoicePlaybackFromFile.ercRet != 0)
       return(rqTsVoicePlaybackFromFile.ercRet);
    fPlaybackFinished = TRUE; /* indicate respond rcvd */
    return(ercOK);
}
/*******************************************************/
int ReadTouchToneRespond()
{
    /*
     * Check request block for DTMF char or timeout.
     */
    switch(rqTsReadTouchTone.ercRet) {
       case ercOK:
          fDTMFReceived = TRUE;
          break;
       case lercDTMFTimeout:
          fDTMFTimeout = TRUE;
          break;
```

Example 14-3. Response.c (cont.)

```
      default:
         return(rqTsReadTouchTone.ercRet);
   }
   return(ercOK);
}
/*******************************************************/
int CleanupLine()
{
   int erc;
   erc = TsOnHook(iVpModule, (iLine - 1)); /* hangup */
   if (erc != ercOK) return erc;
   erc = TsDoFunction(iVpModule, 13); /* unlock codec */
   return erc ;
}
/*******************************************************/
ErrorCleanup(erc)
int erc;
{
   if (erc != ercOK) { /* cleanup as much as possible */
      TsVoiceStop(iVpModule);
      CleanupLine();
      CloseFile(fhInitial);
      CloseFile(fhDTMF1);
      CloseFile(fhDTMF2);
      CloseFile(fhError);
      ErrorExit(erc);
   }
}
/*******************************************************/
int PlayMessage(fh)
int fh;
{
   erc = SetupVoiceFileVCSForPlayback(fh);
   if (erc != ercOK) return erc;
   erc = SetupVoiceFileRqForPlayback();
   if (erc != ercOK) return erc;
   erc = SetupReadTouchToneRq(); /* Setup DTMF rq */
   if (erc != ercOK) return erc;
   fPlaybackFinished = FALSE; /* Reset state flags */
   fDTMFReceived = FALSE;
```

Example 14-3. Response.c (cont.)

```
    /*
     * Issue the requests.
     */
    erc = Request(&rqTsVoicePlaybackFromFile);
    if (erc != ercOK) return erc;
    erc = Request(&rqTsReadTouchTone);
    return erc;
}
/*********************************************************/
int MapDTMFtoFH(ch)
char ch;
{
    switch(ch) {
        case '1':
            return fhDTMF1;
        case '2':
            return fhDTMF2;
        case '#':
            return fhDTMFPound;
        default:
            return fhError;
    }
}
/*********************************************************/
int ConnectVoice()
{
    if (iLine == lLine1) {
        erc = TsVoiceConnect(iVpModule,FALSE,TRUE,FALSE);
        return erc;
    } else if (iLine == lLine2) {
        erc = TsVoiceConnect(iVpModule,FALSE,FALSE,TRUE);
        return erc;
    } else return(lercInvalidParameters);
}
```

Example 14-4: Voice Memory Playback

This program takes between one and three files as arguments, concatenates them in memory, and then plays them when the telephone unit is taken offhook.

The Voice File Header from the first file is used to initialize the fields of the Voice Control Structure. Any subsequent voice files must have been recorded compatibly, or an error is returned.

See the section above, "Voice Playback from Memory," for a review of the steps necessary to playback a voice file from memory rather than from disk. This program has been written so that the Append procedure is general enough to be called for any number of voice files, as long as the limit of 110 voice sectors is not exceeded.

Example 14-4. Memory.c

```
/*
 * Program title:  Memory.c
 * Compiler:  Metaware High C Compiler
 *
 * Memory
 *  File
 *  [File]
 *  [File]
 *
 *
 * Voice Control Structure
 */
struct VCSType {
   int fh;
   long lfaStart;
   long lfaMax;
   long qSampleStart;
   long qSampleMax;
   int cPauseMax;
   int cSampleOn;
   int cSampleOff;
   char f6KHz;
   char fAutoStart;
   char fNoPause;
   char fStopOnDialTone;
```

Example 14-4. Memory.c (cont.)

```
    char fAltConnection;
    int nSectorStatusUpdate;
    int sPauseGap;
    char fRawData;
};
/*
 * Voice File Header Entry Structure
 */
struct VDFHEntryType {
    char f6KHz;
    long lfaStart;
    long lfaMax;
    long qSampleStart;
    long qSampleMax;
    char bReserved;
    long dateTime;
    char line;
    char fNoPause;
    char fAltConnection;
    char rgReserved[7];
};

/*
 * Voice File Header Structure
 */
struct VDFHType {
    int signature;
    int version;
    int wReserved;
    struct VDFHEntryType message[15];
    char rgReserved[26];
};

/*
 * LineState is part of TSStatusType
 */
struct LineState {
    char status;
    char fDialing;
    char dialState;
    char fRinging;
```

Example 14-4. Memory.c (cont.)

```
    char iRing;
    char fRingThrough;
    char fOffHook;
    char fHold;
    char fCodec;
    char fDtmfRec;
    int handle;
    char reserved;
    char fModem;
    char reserved2[2];
};
/*
 * TSStatus Structure
 *    Information returned by the TSGetStatus call
 */
struct TSStatusType {
    int iEvent;
    char defaultLine;
    char fNeedCodecConnection;
    char fCodecInUse;
    char aTmcb[3];
    int reserved;
    int baudrate;
    char originateMode;
    char parityMode;
    char lineControlMode;
    char reserved2;
    /*
     * the following are part of tUnitState
     */
    char fOffHook;
    char rgfTLine[2];
    char rgfRingThrough[2];
    char fCodec;
    char fDtmfRec;
    char fMonitor;
    int handle;
    char hookThroughMode;
    char reserved3[5];
```

Example 14-4. Memory.c (cont.)

```
    /*
     * end of tUnitState
     */
    struct LineState rgLineState[2];

    /*
     * codec state
     */
    long lfaCurrent;
    long qSampleCurrent;
    char reserved4[8];
    /*
     * end of codec state
     */
};
/*
 * This union is used to change the offset of pWorkArea
 * as the files are Read into memory from disk file.
 */
union p_u {
    char *p;
    int i[2];
};

#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define Syslit
#define sdType
#include <CTOSTypes.h>

/*
 * CTOS.Lib calls
 */
#define AllocMemorySL
#define CheckErc
#define CloseFile
#define DeallocMemorySL
```

Example 14-4. Memory.c (cont.)

```
#define ErrorExit
#define OpenFile
#define Read
#define RgParam
#include <CTOSLib.h>

/*
 * Function prototypes for Telephone Service requests
 */
pragma Calling_convention(CTOS_CALLING_CONVENTIONS);

int TsDoFunction(int liVpModule, int function);
int TsGetStatus(int liVpModule, void *pStatusRet,
      int sStatusRetMax, int fNoWait);
int TsOnHook(int liVpModule, int line);
int TsVoiceConnect(int liVpModule, char fVoiceUnit,
      char fLine0, char fLine1);
int TsVoicePlaybackFromFile(int liVpModule,
      void *pWorkArea, int sWorkArea, void *pVoiceControl,
      int sVoiceControl, void *pLfaLast, void *pqSampleLast,
      void *pStatusRet);

pragma Calling_convention();
/*
 * Global constants
 */
#define liVpModule 1
#define liLine 0
#define lercVoiceFileTooLarge 1
#define lercNotAVoiceFile 2
#define lercVoiceFilesNotCompatible 3
#define lsWorkAreaMax 64512
#define lercParameterFound 0
#define lercParameterNotFound 2450
#define lRateNotSet 42 /* some magic number */
#define lercNotOffHook 11204
```

Example 14-4. Memory.c (cont.)

```
/*
 * Function definitions
 */
int GetParams();
void CleanUp();
int SetupVCS();

void main()
{
   char fPrompted = FALSE, *pWorkArea;
   int erc, fh1, fh2, fh3, StatusRet;
   struct VCSType VCS;
   struct TSStatusType TSStatus;
   /*
    * sWorkArea is an unsigned long int instead of an
    * int because of overflow when checking for file
    * sizes greater than the allowed buffer size.
    */
   unsigned long int lfaNext, qSampleNext, sWorkArea =
       64512;.
   CheckErc(AllocMemorySL((unsigned int) sWorkArea,
      &pWorkArea));
   erc = GetParams(&fh1, &fh2, &fh3);
   if (erc != ercOK) CleanUp(erc, fh1, fh2, fh3,
       pWorkArea);
   /*
    * AllocMemorySL has been called to allocate a 64K
    * memory segment.  As the voice files are appended
    * to this segment, their length will be added to
    * sWorkArea, which starts out at 8192, as per the
    * instructions in the 2.0 Voice/Data Services
    * Release Notes.
    */
   sWorkArea = 8192;
   erc = SetupVCS(&VCS);
   if (erc != ercOK) CleanUp(erc, fh1, fh2, fh3,
      pWorkArea);
   erc = Append(fh1, &VCS, &sWorkArea,
      pWorkArea);
   if (erc != ercOK) CleanUp(erc, fh1, fh2, fh3,
      pWorkArea);
```

Example 14-4. Memory.c (cont.)

```
if (fh2 != 0) {
   erc = Append(fh2, &VCS, &sWorkArea,
      pWorkArea);
   if (erc != ercOK) CleanUp(erc, fh1, fh2, fh3,
      pWorkArea);
}
if (fh3 != 0) {
   erc = Append(fh3, &VCS, &sWorkArea,
      pWorkArea);
   if (erc != ercOK) CleanUp(erc, fh1, fh2, fh3,
      pWorkArea);
}
/*
 * Last field to set up.  The sWorkArea is not known
 * until after the calls to Append.
 */
VCS.lfaMax = sWorkArea - 8192;
/*
 *  See if telephone unit is connected to liLine.  If
 *  not,then prompt the user and wait until it is.
 */
do {
   CheckErc(TsGetStatus(liVpModule, &TSStatus,
    sizeof(struct TSStatusType), FALSE));
   if (!fPrompted && (TSStatus.fOffHook == FALSE)) {
      printf("Please pick up the handset...\n");
      fPrompted = TRUE;
   }
} while (TSStatus.fOffHook == FALSE);
/*
 * Hang up the line to get rid of dial tone.
 * If the line is already onhook then erc 11204 is
 * returned.  This case is trapped for.
 */
erc = TsOnHook(liVpModule, liLine);
if (!((erc == ercOK) || (erc == lercNotOffHook)))
   CleanUp(erc, fh1, fh2, fh3, pWorkArea);
```

Example 14-4. Memory.c (cont.)

```
   /*
    * Lock the codec
    */
   erc = TsDoFunction(liVpModule, 12);
   if (erc != ercOK) CleanUp(erc, fh1, fh2, fh3,
      pWorkArea);
   erc = TsVoiceConnect(liVpModule, TRUE, FALSE, FALSE);
   if (erc != ercOK) CleanUp(erc, fh1, fh2, fh3,
      pWorkArea);
   erc = TsVoicePlaybackFromFile(liVpModule, pWorkArea,
      (unsigned int) sWorkArea, &VCS,
      sizeof(struct VCSType), &lfaNext, &qSampleNext,
      &StatusRet);
   /*
    * Exit, either normally or abnormally, based on erc.
    */
   CleanUp(erc, fh1, fh2, fh3, pWorkArea);
}
/*****************************************************/
int GetParams(f1, f2, f3)

int *f1, *f2, *f3;
{
   sdType sdParam;
   int erc;

   erc = RgParam(1, 0, &sdParam);
   if (erc != lercParameterFound) {
      return(erc);
   } else {
      erc = OpenFile(f1, sdParam.pb, sdParam.cb,
         "", 0, modeRead);
      if (erc != ercOK) return(erc);
   }
   erc = RgParam(2, 0, &sdParam);
   switch(erc) {
      case lercParameterFound:
         erc = OpenFile(f2, sdParam.pb, sdParam.cb,
            "", 0, modeRead);
         if (erc != ercOK) return(erc);
         break;
```

Example 14-4. Memory.c (cont.)

```
      case lercParameterNotFound:
         *f2 = 0;
         erc = ercOK;
         break;
      default:
         return(erc);
   }
   erc = RgParam(3, 0, &sdParam);
   switch(erc) {
      case lercParameterFound:
         erc = OpenFile(f3, sdParam.pb, sdParam.cb,
            "", 0, modeRead);
         if (erc != ercOK) return(erc);
         break;
      case lercParameterNotFound:
         *f3 = 0;
         erc = ercOK;
         break;
      default:
         return(erc);
   }
   return(ercOK);
}
/*******************************************************/
int SetupVCS(pVCS)

struct VCSType *pVCS;
{
   /*
    * Set up VCS for TsVoicePlaybackFromFile call setting
    * VCS.fh to 0xFFFF causes Telephone Service to playback
    * from memory.
    */
   pVCS->fh = 0xFFFF;
   pVCS->lfaStart = 0;
   pVCS->qSampleStart = 0;
   pVCS->qSampleMax = 0xFFFFFFFF;
   pVCS->cPauseMax = 0;
   pVCS->cSampleOn = 0;
   pVCS->cSampleOff = 0;
```

Example 14-4. Memory.c (cont.)

```
    pVCS->f6KHz = lRateNotSet;
    pVCS->fAutoStart = 0xFF;
    pVCS->fNoPause = 0xFF;
    pVCS->fStopOnDialTone = 0;
    pVCS->fAltConnection = 0;
    pVCS->nSectorStatusUpdate = 0;
    pVCS->sPauseGap = 0;
    pVCS->fRawData = 0;

    return(ercOK);
}
/*******************************************************/
int Append(fh, pVCS, psWorkArea, pWorkArea)
int fh;
struct VCSType *pVCS;
unsigned long int *psWorkArea;
union p_u pWorkArea;
{
    struct VDFHType VDFH;
    int erc, sDataRet;
    /*
     * Read in the first sector of the file
     */
    erc = Read(fh, &VDFH, 512, 0, &sDataRet);
    if (erc != ercOK) return(erc);

    /*
     * Will the file fit in the workarea?
     */
    if ((VDFH.message[0].lfaMax -
         VDFH.message[0].lfaStart +
         *psWorkArea) > lsWorkAreaMax)
       return (lercVoiceFileTooLarge);
    /*
     * Check to see if this file is recorded at same rate
     * as previous file.  If VCS.f6KHz is set to
     *lRateNotSet, then set rate.  Otherwise, compare
     *rates.
     */
```

Example 14-4. Memory.c (cont.)

```
    if (pVCS->f6KHz == lRateNotSet) {
       pVCS->f6KHz = VDFH.message[0].f6KHz;
    } else {
       if (pVCS->f6KHz != VDFH.message[0].f6KHz) {
          return(lercVoiceFilesNotCompatible);
       }
    }
    /*
     * Munge pointer so that the voice records are
     * read in after either the 8192 byte header or the
     * previous file.
     */
    pWorkArea.i[0] = (unsigned int) *psWorkArea;
    erc = Read(fh, pWorkArea.p,
       (VDFH.message[0].lfaMax- VDFH.message[0].lfaStart),
       512, &sDataRet);
    if (erc != ercOK) return(erc);
    /*
     * Update sWorkArea to include this file
     */
    *psWorkArea = VDFH.message[0].lfaMax -
                  VDFH.message[0].lfaStart +
                  *psWorkArea;
    return(ercOK);
}
/*******************************************************/
void CleanUp(erc, fh1, fh2, fh3, pWorkArea)
int erc, fh1, fh2, fh3;
char *pWorkArea;
{
    /*
     * Close the files, if they are open.  Error checking
     * is not done, because we want to clean up as much
     * as possible before leaving.
     */
    if (fh1 != 0) CloseFile(fh1);
    if (fh2 != 0) CloseFile(fh2);
    if (fh3 != 0) CloseFile(fh3);
```

Example 14-4. Memory.c (cont.)

```
    /*
     * Unlock the codec
     */
    TsDoFunction(liVpModule, 13);
    /*
     * Deallocate SL memory
     */
    DeallocMemorySL(pWorkArea, lsWorkAreaMax);
    /*
     * Handle the status code
     */
    switch (erc) {
       case lercVoiceFileTooLarge:
          printf("The voice files are larger than \
                 56K.\n");
          ErrorExit(0);
          break;
       case lercNotAVoiceFile:
          printf("The specified voice file is \
                 invalid.\n");
          ErrorExit(0);
          break;
       case lercVoiceFilesNotCompatible:
          printf("Voice file recorded rates \
                 different.\n");
          ErrorExit(0);
          break;
       default:
          ErrorExit(erc);
    }
}
```

Example 14-5: Data Call

This program shows how a telephone or voice application can make a data call. It uses the Asynchronous Terminal Emulator (ATE). The exit run file for the partition is saved in the Application System Control Block (ASCB), and the currently executing run file (*DataCall.run*) is substituted as the exit run file before the Chain to ATE. When ATE terminates, *DataCall.run* gets reloaded, and if it detects that it is called because of the termination of ATE, it restores the previous run file and then exits. This example does this because an application that places a data call would probably want to resume after completion of the call. See the comments in the code for the entry and exit points after running such a program from a larger application.

Example 14-5. DataCall.c

```
/*
 * Program title:  DataCall.c
 * Compiler:  Metaware High C Compiler
 *
 * DataCall
 *  Comm Channel
 *  Baud rate
 *  Stop bits
 *  Parity
 *  Data bits
 *  XON/XOFF
 *  Script
 *  Open mode
 *  [Dial String]
 *
 *  Explanation of command form options:
 *
 *  Comm Channel:      Comm Port or VP Line
 *                     (default = [Phone]1)
 *
 *  Baud rate:         Data transmission speed
 *                     (default = 1200)
 *
 *  Stop bits:         Number of data stop bits per frame
 *                     (default = 1)
 *
```

Example 14-5. DataCall.c (cont.)

```
 *  Parity:             Definition of frame parity bit.
 *                      (default = Even)
 *  XON/XOFF:           Use XON/XOFF flow control?
 *                      (default = no)
 *
 *  Script:             Filename of command script.
 *                      (default = "")
 *
 *  Open mode:          Default = Dial
 *
 *  Dial string:        Phone number to dial (required)
 */
#include <string.h>
#include <stdlib.h>
#define Syslit
#define sdType
#include <CTOSTypes.h>
#define sAscb 304

struct ASCBType {
   int fhSwapFile;
   char *pVLPB;
   char fExecScreen;
   char fChkBoot;
   int ercRet;
   char *pbMsgRet;
   int cbMsgRet;
   char DtModeID;
   char DtModeYmd;
   char DtModeTim;
   char DtModeFmt;
   char reserved;
   char fChainedTo;
   char fTermination;
   char fVacate;
   int oLastTask;
   char fExecFont;
   char bActionCode;
   int cParMemArray;
   int ALSignature;
   int fhContext;
```

Example 14-5. DataCall.c (cont.)

```
    char fDollarContext;
    char *pExitRunFileBuf;
    char *pbPassedData;
    int cbPassedData;
    char sbNodeMail[13];
    long qMailId;
    int naMailServer;
    char sbUserName[31];
    char sbUserPswd[13];
    char sbCmdFile[79];
    char cbExitRunFile;
    char ExitRunFile[78];
    char cbPswd;
    char Pswd[12];
    char priority;
    char fColor;
    char rgbColorBytes[8];
    char fReverseVideo;
    char fBackGround;
    char bBackGround;
    char fFilter;
    char bStatusRet;
    char reserved2[10];
}
;struct ExParDescType {
    char sbCurRunFileSpec[79];
    char sbExitRunFileSpec[79];
    char sbExitRunFilePswd[13];
    char ExitRunFilePriority;
};
#define AllocMemoryLL
#define Chain
#define CheckErc
#define CloseFile
#define ErrorExit
#define GetpASCB
#define GetpStructure
#define OpenFile
#define QueryExitRunFile
#define ResetMemoryLL
#define RgParam
```

Example 14-5. DataCall.c (cont.)

```
#define RgParamInit
#define RgParamSetSimple
#define SetExitRunFile
#include <CTOSLib.h>
/*
 * constants
 */
#define lCommChannel 2
#define lBaudRate 3
#define lStopBits 4
#define lParity 5
#define lDataBits 6
#define lXOnXOff 11
#define lScript 12
#define lOpenMode 13
#define lDialChars 15
#define lVLPBSize 300
#define lFileNameMax 30
#define lPasswordMax 12
#define lCTSignature 0x4354
#define lPriority 0x40
#define lParameterFound 0
#define lParameterNotFound 2450
/*
 * procedure definitions
 */
int GetParams();
int SaveExitRunFile();
int RestoreExitRunFile();
/*
 * global variables
 */
char rgbParams[9][20];

/*******************************************************/
void main()
{
   int fh;
   char *pVLPB;
   struct ASCBType *pASCB;
   struct ExParDescType *pEPD;
```

Example 14-5. DataCall.c (cont.)

```
sdType sdTemp;
/*
 * Check to see if we are returning from the Chain
 */
CheckErc(GetpASCB(&pASCB));
/*
 * In SaveExitRunFile the ALSignature is set to CT
 * so we know if we are returning from a Chain.
 */
if (pASCB->ALSignature == lCTSignature) {
   CheckErc(RestoreExitRunFile());
   /*
    * If this were an actual application, this is
    * where the application would continue after the
    * data call.  For this example, we exit.
    */
   ErrorExit(0);
}
/*
 * Save all the parameters to DataCall before
 * calling RgParamInit
 */
CheckErc(GetParams());
/*
 * Prepare VLPB for ATE command options.
 */
CheckErc(ResetMemoryLL());
CheckErc(AllocMemoryLL(lVLPBSize, &pVLPB));
CheckErc(RgParamInit(pVLPB, lVLPBSize, 17));
/*
 * Fill in fields of VLPB.
 */
sdTemp.pb = &rgbParams[0][1];
sdTemp.cb = rgbParams[0][0];
if (sdTemp.cb != 0)
   CheckErc(RgParamSetSimple(lCommChannel, &sdTemp));
sdTemp.pb = &rgbParams[1][1];
sdTemp.cb = rgbParams[1][0];
if (sdTemp.cb != 0)
   CheckErc(RgParamSetSimple(lBaudRate, &sdTemp));
sdTemp.pb = &rgbParams[2][1];
```

Example 14-5. DataCall.c (cont.)

```
sdTemp.cb = rgbParams[2][0];
if (sdTemp.cb != 0)
   CheckErc(RgParamSetSimple(lStopBits, &sdTemp));
sdTemp.pb = &rgbParams[3][1];
sdTemp.cb = rgbParams[3][0];
if (sdTemp.cb != 0)
   CheckErc(RgParamSetSimple(lParity, &sdTemp));
sdTemp.pb = &rgbParams[4][1];
sdTemp.cb = rgbParams[4][0];
if (sdTemp.cb != 0)
   CheckErc(RgParamSetSimple(lDataBits, &sdTemp));
sdTemp.pb = &rgbParams[5][1];
sdTemp.cb = rgbParams[5][0];
if (sdTemp.cb != 0)
   CheckErc(RgParamSetSimple(lXOnXOff, &sdTemp));
sdTemp.pb = &rgbParams[6][1];
sdTemp.cb = rgbParams[6][0];
if (sdTemp.cb != 0)
   CheckErc(RgParamSetSimple(lScript, &sdTemp));
sdTemp.pb = &rgbParams[7][1];
sdTemp.cb = rgbParams[7][0];
if (sdTemp.cb != 0)
   CheckErc(RgParamSetSimple(lOpenMode, &sdTemp));
sdTemp.pb = &rgbParams[8][1];
sdTemp.cb = rgbParams[8][0];
if (sdTemp.cb != 0)
   CheckErc(RgParamSetSimple(lDialChars, &sdTemp));
/*
 * Check to make sure that the ATE run file is present.
 */
CheckErc(OpenFile(&fh, "[sys]<sys>ATE.run", 17, "",
   0, modeRead));
CheckErc(CloseFile(fh));
/*
 * Save the current exit run file before setting it
 * to this program.
 */
CheckErc(SaveExitRunFile());
```

Example 14-5. DataCall.c (cont.)

```
    /*
     * Get address of the Extended Partition Descriptor
     */

    CheckErc(GetpStructure(0, 0, &pEPD));
    /*
     * Set exit run file to ourselves.
     */
    CheckErc(SetExitRunFile(&(pEPD->sbCurRunFileSpec[1]),
       pEPD->sbCurRunFileSpec[0],
       &(pEPD->sbExitRunFilePswd[1]),
       pEPD->sbExitRunFilePswd[0],
       pEPD->ExitRunFilePriority));
    /*
     * Chain to ATE.
     */
    CheckErc(Chain("[sys]<sys>ATE.run", 17, 0, 0,
             lPriority, 0, 0));
}
/*******************************************************/
int GetParams()

{
    /*
     * rgbParams is a two dimensional array that holds
     * the field values from the command form of DataCall
     */
    int erc;
    sdType sdParam;

    /*
     * Get the parameters, filling in the default values
     * if they are not supplied.
     */
    erc = RgParam(1, 0, &sdParam);
    switch(erc) {
       case lParameterFound:
          strncpy(&rgbParams[0][1], sdParam.pb, sdParam.cb);
          rgbParams[0][0] = sdParam.cb;
          break;
```

Example 14-5. DataCall.c (cont.)

```
    case lParameterNotFound:
        strncpy(&rgbParams[0][1], "[Phone]1", 8);
        rgbParams[0][0] = 8;
        break;
    default:
        return(erc);
}
erc = RgParam(2, 0, &sdParam);
switch(erc) {
    case lParameterFound:
        strncpy(&rgbParams[1][1], sdParam.pb, sdParam.cb);
        rgbParams[1][0] = sdParam.cb;
        break;
    case lParameterNotFound:
        strncpy(&rgbParams[1][1], "1200", 4);
        rgbParams[1][0] = 4;
        break;
    default:
        return(erc);
}
erc = RgParam(3, 0, &sdParam);
switch(erc) {
    case lParameterFound:
        strncpy(&rgbParams[2][1], sdParam.pb, sdParam.cb);
        rgbParams[2][0] = sdParam.cb;
        break;
    case lParameterNotFound:
        strncpy(&rgbParams[2][1], "1", 1);
        rgbParams[2][0] = 1;
        break;
    default:
        return(erc);
}
erc = RgParam(4, 0, &sdParam); switch(erc) {
    case lParameterFound:
        strncpy(&rgbParams[3][1], sdParam.pb, sdParam.cb);
        rgbParams[3][0] = sdParam.cb;
        break;
```

Example 14-5. DataCall.c (cont.)

```
    case lParameterNotFound:
        strncpy(&rgbParams[3][1], "Even", 4);
        rgbParams[3][0] = 4;
        break;
    default:
        return(erc);
}

erc = RgParam(5, 0, &sdParam);
switch(erc) {
    case lParameterFound:
        strncpy(&rgbParams[4][1], sdParam.pb, sdParam.cb);
        rgbParams[4][0] = sdParam.cb;
        break;
    case lParameterNotFound:
        strncpy(&rgbParams[4][1], "7", 1);
        rgbParams[4][0] = 1;
        break;
    default:
        return(erc);
}
erc = RgParam(6, 0, &sdParam);
switch(erc) {
    case lParameterFound:
        strncpy(&rgbParams[5][1], sdParam.pb, sdParam.cb);
        rgbParams[5][0] = sdParam.cb;
        break;
    case lParameterNotFound:
      strncpy(&rgbParams[5][1], "no", 2);
        rgbParams[5][0] = 2;
        break;
    default:
        return(erc);
}

erc = RgParam(7, 0, &sdParam);
switch(erc) {
    case lParameterFound:
        strncpy(&rgbParams[6][1], sdParam.pb, sdParam.cb);
        rgbParams[6][0] = sdParam.cb;
```

Example 14-5. DataCall.c (cont.)

```
        case lParameterNotFound:
            strncpy(&rgbParams[6][1], "", 0);
            rgbParams[6][0] = 0;
            break;
        default:
            return(erc);
    }

    erc = RgParam(8, 0, &sdParam);
    switch(erc) {
        case lParameterFound:
            strncpy(&rgbParams[7][1], sdParam.pb, sdParam.cb);
            rgbParams[7][0] = sdParam.cb;
            break;
        case lParameterNotFound:
            strncpy(&rgbParams[7][1], "Dial", 4);
            rgbParams[7][0] = 4;
            break;
        default:
            return(erc);
    }
    /*
     * The dial string must be specified, so if it is not
     * present, the program will call ErrorExit
     */
    erc = RgParam(9, 0, &sdParam);
    switch(erc) {
        case lParameterFound:
            strncpy(&rgbParams[8][1], sdParam.pb, sdParam.cb);
            rgbParams[8][0] = sdParam.cb;
            break;
        default:
            return(erc);
    }
    return(ercOK);
}
```

Example 14-5. DataCall.c (cont.)

```
/*******************************************************/
int SaveExitRunFile()
{
   char sbExitRunFile[lFileNameMax],
      sbExitRunFilePswd[lPasswordMax], bExitRunFilePr;
   int erc;
   struct ASCBType *pASCB;
   /*
    * Get address of ASCB
    */
   erc = GetpASCB(&pASCB);
   if (erc != ercOK) return(erc);
   /*
    * Save exit run file.
    */
   erc = QueryExitRunFile(&sbExitRunFile, lFileNameMax,
      &sbExitRunFilePswd, lPasswordMax,
      &bExitRunFilePr);
   if (erc != ercOK) return(erc);
   /*
    * Update ASCB with run file name
    */
   pASCB->cbExitRunFile = sbExitRunFile[0];
   strncpy(pASCB->ExitRunFile, &sbExitRunFile[1],
      sbExitRunFile[0]);
   pASCB->cbPswd = sbExitRunFilePswd[0];
   strncpy(pASCB->Pswd, &sbExitRunFilePswd[1],
      sbExitRunFilePswd[0]);
   pASCB->priority = bExitRunFilePr;
   /*
    * Use batch signature to know if we are returning
    * from Chain.
    */
   pASCB->ALSignature = lCTSignature;
   return(ercOK);
}
```

Example 14-5. DataCall.c (cont.)

```
int RestoreExitRunFile()
{
   int erc;
   struct ASCBType *pASCB;
   /*
    * Get address of ASCB
    */
   erc = GetpASCB(&pASCB);
   if (erc != ercOK) return(erc);
   /*>
    * Restore original exit run file so that DataCall
    * exits normally.
    */
   erc = SetExitRunFile(pASCB->ExitRunFile,
      pASCB->cbExitRunFile, pASCB->Pswd, pASCB->cbPswd,
      pASCB->priority);
   if (erc != ercOK) return(erc);
   pASCB->cbExitRunFile = 0;
   /*
    * Reset signature
    */
   pASCB->ALSignature = 0;
   return(ercOK);
}
```

Data Structures

The following data structures are associated with Voice/Data Services.

Table 14-2. Telephone Service Configuration File Format

Offset	Field	Size (bytes)	Description
0	*signature*	2	Always 'tC'
2	*version*	2	Configuration file version number
4	*TsConfig*	252	Configuration information
256	*OperatorConfig*	768	Used for storing operator configuration

signature

Is always 'tC'.

version

Is the version number of the configuration file (currently 1).

TsConfig

Is configuration information. For the format of this information, see Table 14-3, in this section.

operatorConfig

Is an area used by the Operator program to store configuration information, such as the phone number of each line, diaing prefixes, and so forth.

Table 14-3. Telephone Service Configuration Block

Offset	Field	Size (bytes)	Description
0	*rgDtmfGenOff*	2	Time between DTMF tones
2	*rgDtmfGenOn*	2	Time during DTMF tones
4	*rgFlashTime*	2	Length of a default flash ("@" in dial string)
6	*rgPauseTime*	2	Length of a default pause ("~" in dial string)
8	*rgfPulseDial*	2	TRUE means pulse dialing is used
10	*rgRingHz*	4	Workstation monitor ring signal frequency
14	*rgiRingThrough*	2	Ring number in which the ring current sent
16	*rgRingMode*	2	Ring mode for each line
18	*rgCodecMode*	2	CODEC mode
20	*nSecHoldRing*	2	Ring if on hold this many seconds
22	*nSecHoldHangup*	2	Hang up if on hold this many seconds
24	*rgbConfigfile*	100	Telephone Service configuration file name
124	*rgAction*	64	Action key values mapped to TsDoFunction
188	*rgFunction*	64	Functions executed when action key typed

rgDtmfGenOff
rgDtmfGenOn

Specify the time in units of 10 ms, between (*rgDtmfGenOff*) and during (*rgDtmfGenOn*) DTMF tones when dialing. Most PBXs can handle values of 10 (100ms) and some as little as 6 (60ms). The default is 6 for *rgDtmfGenOff* and 8 for *rgDtmfGenOn*.

rgFlashTime

Specifies the length of a default flash ("@" character in a dial string) in units of 100ms. Most PBXs use a value of 10 (1 second). The default is 10.

rgPauseTime

Specifies the length of a default pause ("~" character in a dial string) in units of 100ms. The default is 20.

rgfPulseDial

Is a flag. If TRUE, pulse dialing is used instead of DTMF generation. The default is FALSE.

rgRingHz

Is an array used to specify the frequency of the workstation monitor's ringing signal when the telephone line is ringing. A value of 0 means no tone is generated. The range of frequencies is 1 to 255.

Byte	Description
0	neither line (default 0)
1	line 1 (default 40)
2	line 2 (default 90)
3	both lines 1 and 2 (default 150)

rgiRingThrough

Is the ring number in which the ring current will be sent directly to the telephone unit. A value of 0FFFFh means the ring current is never passed through. The default is 4.

rgRingMode

Is the ring mode for each line. When a line rings, some combination of monitor ringing and telephone unit ringing is done. The ring mode values are as follows:

Value	Description
0	Do not ring either the telephone unit or the monitor.
1	Ring only the telephone unit.
2	Ring only the monitor.
3	Always ring both the telephone unit and the monitor.
4	Ring the monitor for the number of rings specified in *rgiRingThrough* above, then ring the telephone unit.
5	Ring the monitor for the number of rings specified in *rgiRingThrough* above, then ring both the monitor and the telephone unit (default).

rgCodecMode

Is the CODEC mode (always 0).

nSecHoldRing

Ring if on hold this many seconds (default is 300).

nSecHoldHangup

Hang up if on hold this many seconds (default is 600).

rgbConfigfile

Is the Telephone Service configuration file name.

rgAction

Is an array of action key values (encoded keyboard values) to be mapped to TsDoFunction functions. A value of 0 terminates the list.

rgFunction

Is an array of functions (see TsDoFunction in the *CTOS Procedural Interface Reference Manual*) to be executed when the corresponding action key listed in *rgAction* is typed.

Table 14-4. Telephone Status Structure

Offset	Field	Size (bytes)	Description
0	*iEvent*	2	Incremented with status structure change
2	*defaultLine*	1	Default line as selected by TsDoFunction
3	*fNeedCodecConnection*	1	TRUE means a voice operation is waiting
4	*fCodecInUse*	1	CODEC is recording or playing back
5	*pVPCB*	3	24-bit physical address of the VPCB
8	reserved	2	
10	*baudRate*	2	Either 300 or 1200
12	*originateMode*	1	Either TRUE or FALSE
13	*parityMode*	1	Is 0, 1, 2, 3, or 4
14	*lineControlMode*	1	Either 0 or 1
15	reserved	1	
16	*tUnitState*	16	Telephone unit state (see tUnitState below)
32	*rgLineState*(2)	32	Line 1/line 2 states (see lineState below)
64	*codecState*	16	CODEC state (see codecState below)

iEvent

Is a counter that is incremented every time the contents of the status structure changes.

defaultLine

Is the default line, as selected by TsDoFunction.

fNeedCodecConnection

Is a flag that is TRUE if a voice operation is waiting for a connection.

fCodecInUse

Is a flag that is TRUE if the CODEC is recording or playing back voice.

pVPCB

Is the 24-bit physical address of the Voice Processor Control Block (VPCB).

baudRate

Is either 300 or 1200.

originateMode

Is TRUE if the originator or FALSE if the answerer.

parityMode

Is the parity control, where the values are as follows:

Value	Description
0	none
1	even (bit 7 is set or cleared so that there are always an even number of bits set in each byte)
2	odd (bit 7 is set or cleared so that there are always an odd number of bits set in each byte)
3	1 (bit 7 is always set); also known as 'mark'
4	0 (bit 7 is always cleared); also known as 'space'

lineControlMode

Is one of the following values:

Value	Description
0	no flow control
1	XON/XOFF flow control

tUnitState

Is the telephone unit state. (See Table 14-5.)

rgLineState

Are the line 1 and line 2 states. (See Table 14-6.)

codecState

Is the CODEC state. (See Table 14-7.)

Table 14-5. tUnitState Fields of the Telephone Status Structure

Offset	Field	Size (bytes)	Description
0	*fOffHook*	1	TRUE means the telephone unit is offhook
1	*rgfTLine(2)*	2	Line 1/line 2 connection to telephone unit
3	*rgfRingThrough(2)*	2	Line 1/line 2 ring voltage to telephone unit
5	*fCodec*	1	Telephone unit connection to the CODEC
6	*fDtmfRec*	1	Telephone unit DTMF decoder connection
7	*fMonitor*	1	Telephone unit monitor mode connection
8	*handle*	2	Telephone unit connection handle
10	*hookThroughMode*	1	Mode for passing through on/offhook
11	reserved	5	

fOffHook

Is a flag that is TRUE if the telephone unit is offhook.

rgfTLine

Is an array of flags which, if TRUE, mean line 1 and/or line 2 is connected to the telephone unit.

rgfRingThrough

Is an array of flags which, if TRUE, mean line 1 or line 2 is connected directly to the telephone unit to allow ring voltage to be passed through.

fCodec

Is a flag that is TRUE if the telephone unit is connected to the CODEC.

fDtmfRec

Is a flag that is TRUE if the telephone unit is connected to the DTMF decoder.

fMonitor

Is a flag that is TRUE if the telephone unit is connected in monitor mode.

handle

Is the handle associated with the current telephone unit connection.

hookThroughMode

Is the mode for passing through on/off hook from the telephone unit to the telephone lines. The values are as follows:

Value	Description
0	neither line
1	line 1
2	line 2
3	both lines

Table 14-6. lineState Fields of the Telephone Status Structure

Offset	Field	Size (bytes)	Description
0	*status*	1	Line status
1	*fDialing*	1	Dial string is being processed on this line
2	*dialState*	1	The current state of dialing
3	*fRinging*	1	If TRUE, the line is ringing
4	*iRing*	1	Number of rings
5	*fRingThrough*	1	Passing ring through to the telephone unit
6	*fOffHook*	1	TRUE means the line is offhook
7	*fHold*	1	TRUE means the line is on hold
8	*fCodec*	*1*	TRUE means the CODEC is being used
9	*fDtmfRec*	1	TRUE means the DTMF detector is being used
10	*handle*	2	Telephone unit connection handle
12	reserved	1	
13	*fModem*	1	TRUE means the modem is being used
14	reserved	2	

status

Is the line status. The values are as follows:

Value	Description
0	onhook
1	ring current detected
2	modem not ready
3	modem ready (FSK, 300 baud)
4	modem ready (PSK, 1200 baud)
5	offhook

fDialing

Is a flag that is TRUE if a dial string is being processed on this line.

dialState

Is the current state of dialing. The values are sa follows:

Value	Description
0	idle
1	generating DTMF
2	analyzing CPTR (waiting for dial tone)
3	generating pulse
4	generating flash
5	generating pause
6	analyzing CPTR (waiting for any tone)
7	analyzing CPTR (waiting for answer)
255	performing error recovery

fRinging

Is a flag that is TRUE if the line is ringing.

iRing

Is the number of rings.

fRingThrough

Is a flag that is TRUE if the ringing is being passed through to the telephone unit.

fOffHook

Is a flag that is TRUE if the line is offhook.

fHold

Is a flag that is TRUE if the line is on hold.

fCodec

Is a flag that is TRUE if the CODEC is being used.

fDtmfRec

Is a flag that is TRUE if the DTMF detector is being used.

handle

Is the connection handle associated with the current telephone unit connection.

fModem

Is a flag that is TRUE if the modem is being used.

Table 14-7. codecState Fields of theTelephone Status Structure

Offset	Field	Size (bytes)	Description
0	*lfaCurrent*	4	Lfa of data record last processed
4	*qSampleCurrent*	4	Data byte number last processed
8	reserved	8	

lfaCurrent

Is the logical file address (lfa) of the data record last processed.

qSampleCurrent

Is the data-byte number last processed.

Table 14-8. Voice File Header

Offset	Field	Size (bytes)	Description
0	*signature*	2	Must be 'VC' (4356h)
2	*version*	2	Version of file
4	*nMessages*	2	Number of recordings in this file
6	*rgMessages*(15)	480	See Table 14-9
485	reserved	26	

signature

Must be 'VC' (4356h).

version

Is the version of the file. For the Telephone Service, this value is 1.

nMessages

Is the number of messages in the file.

rgMessages

Is the array of 15 messages. (See Table 14-9.)

Table 14-9. rgMessages Fields of theVoice File Header

Offset	Field	Size (bytes)	Description
0	*f6KHz*	1	If TRUE, the data was recorded at 6KHz
1	*lfaStart*	4	Starting lfa in file of voice data
5	*lfaMax*	4	Ending lfa in file of voice data
9	*qSampleStart*	4	Starting data byte count of voice data
13	*qSampleMax*	4	Ending data byte count of voice data
17	reserved	1	
18	*dateTime*	4	System date/time when recording made
22	*line*	1	Line over which recording was made
23	*fNoPause*	1	FALSE if pause compressed
24	*fAltConnection*	1	TRUE if recorded with alternate connection
25	reserved	1	
26	*rgbReserved*	6	

f6KHz

Is a flag that is TRUE if the data was recorded at 6KHz.

lfaStart

Is the starting lfa in the file of the recording.

lfaMax

Is the ending lfa in the file of the recording.

qSampleStart

Is the starting data byte count of the recording.

qSampleMax

Is the ending data byte count of the recording.

dateTime

Is the system date and time when the recording was made.

line

Is the line over which the recording was made. The values of line are as follows:

Value	Description
0	telephone unit
1	telephone line 1
2	telephone line 2

fNoPause

Is a flag that is TRUE if the pause compression was suppressed during recording.

fAltConnection

Is a flag that is TRUE if the alternate (amplified) connection was used to record.

Table 14-10. Voice File Record

Offset	Field	Size (bytes)	Description
0	*qSampleStart*	4	Number of the first sample in this record
4	*signature*	1	Always 'V' (56h)
5	*version*	1	Version number of data file
6	*rgbSample*	506	Voice data

qSampleStart

Is the accumulated sample number of the first sample in this record.

signature

Is always 'V' (56h).

version

Is the version number of the data file. The value 1, the only valid value, indicates B25 X-Bus adaptive pulse code modulation (ADPCM)

rgbSample

Is the voice data bytes.

Table 14-11. Voice Control Structure

Offset	Field	Size (bytes)	Description
0	*fh*	2	Is an open file handle
2	*lfaStart*	4	File position to start the recording/playback
6	*lfaMax*	4	Position where record/playback terminate
10	*qSampleStart*	4	Sample number to start record or playback
14	*qSampleMax*	4	Number terminating recording/playback
18	*cPauseMax*	2	Max silence before recording terminated
20	*cSampleOn*	2	Determine the playback "fast forward" rate
22	*cSampleOff*	2	Determine the playback "fast forward" rate
24	*f6KHz*	1	Determines sampling rate
25	*fAutoStart*	1	Determines record/playback start
26	*fNoPause*	1	Disables pause detection hardware
27	*fStopOnDialTone*	1	Recording terminated if dial tone detected
28	*fAltConnection*	1	Alternate connection increases gain levels
29	*nSectorStatusUpdate*	2	Response to TsGetStatus requests
31	*sPauseGap*	2	Number of data bytes in pause
33	*fRawData*	1	Data not checked for escape sequences
34	reserved	1	

fh

Is the open file handle (or 0FFFFh if playing back from memory) of the voice file.

lfaStart

Is the logical file address (lfa) at which to start recording or playing back.

lfaMax

Is the lfa which, if reached, terminates recording or playback.

qSampleStart

Is the sample number to be assigned to the first sample recorded or to be skipped to on playback.

qSampleMax

Is the sample number, which if reached terminates recording or playback.

cPauseMax

Is the maximum silence (in units of 100ms) when recording before the recording is terminated. (The terminating pause is not included in the recording.)

On playback, all pauses are truncated to this amount. A value of 0FFFFh means no pause termination or truncation.

cSampleOn
cSampleOff

Determine the playback "fast forward" rate. A zero value for *cSampleOff* means playback at normal speed. The *cSampleOn* value determines the number of pairs of samples that are played back, and *cSampleOff* is the number of sample pairs that are discarded. *cSampleOn* should be greater than 50. To playback at double speed, values of 100 and 100 are used. To playback at triple speed, values of 100 and 200 are used.

f6KHz

Is a flag that is TRUE if the CODEC sampling rate is to be 6KHz. Otherwise the rate is 8KHz (8000 4-bit samples per second).

fAutoStart

Is a flag that is TRUE if recording or playing back is to start as soon as the telephone unit is placed offhook or immediately if it is already offhook.

fNoPause

Is a flag that is TRUE if the pause detection hardware is to be disabled and no pause detection or compression is to be done.

fStopOnDialTone

Is a flag that is TRUE if the recording is to be terminated if a dial tone is detected.

fAltConnection

Is a flag that is TRUE if an alternate connection is to be used (if possible) to increase gain levels.

nSectorStatusUpdate

Is a value that, if greater than zero, causes any TsGetStatus requests to be responded to each time the specified number of sectors has been processed.

sPauseGap

Is the number of data bytes after start of pause and before end of pause where the actual data is used. If *sPauseGap* is 0, the default value (250) is used.

fRawData

Is a flag that is TRUE if the voice data is to be recorded or played back without examination of the data for escape sequences.

Table 14-12. Data Control Structure

Offset	Field	Size (bytes)	Description
0	*openMode*	1	Data open mode
1	*fOriginate*	1	TRUE programs modem in originate mode
2	*cTimeout*	2	Maximum time to complete the open
4	*baudrate*	2	Either 300 or 1200
6	*parity*	1	Is the parity control
7	*lineControl*	1	Sets flow control mode
8	*fLL*	1	Termination request doesn't close service
9	*fEndOfBlock*	1	bEndOfBlock used as last character
10	*bEndOfBlock*	1	Character terminating TsDataRead
11	*fHangup*	1	TRUE places the line onhook
12	*fCharMode*	1	Data processing on a per character basis
13	*fNoWaitForDialTone*	1	Telephone Service waits for dial tone
14	*fReadTimeoutReset*	1	Counter is reset when character is received
15	*fWriteTimeoutReset*	1	Counter reset when character transmitted

openMode

Is the data open mode. The values are as follows:

Value	Description
0	Convert. Convert an existing voice call to a data call.
1	Accept. Wait for an incoming call to occur.
2	Dial. Place an outgoing data.

fOriginate

Is a flag. If TRUE the modem is the originator. Otherwise it is the answering party.

cTimeout

Is the maximum time that is allowed to complete the open after the initial connection before returning status code 11206 (Timeout).

baudrate

Is either 300 or 1200.

parity

Is the parity control. The values are as follows:

Value	Description
0	none
1	even (bit 7 is set or cleared so that there are always an even number of bits set in each byte)
2	odd (bit 7 is set or cleared so that there are always an odd number of bits set in each byte)
3	1 (bit 7 is always set); also known as 'mark'
4	0 (bit 7 is always cleared); also known as 'space'

lineControl

Is the state of flow control. A value of 0 indicates no flow control. A value of 1 indicates XON/XOFF flow control.

fLL

Is a flag that should be set TRUE by the program if it is running on a single partition version of the operating system.

fEndOfBlock

Is a flag. TRUE means the byte value in bEndOfBlock is to be used as the last character in any TsDataRead operation.

bEndOfBlock

Is the character terminates a TsDataRead operation when fEndOfBlock is TRUE.

fHangup

Is a flag. TRUE means the line is placed onhook at the completion of data transmission. Otherwise it is placed on hold when the data call is closed or terminated.

fCharMode

Is a flag. TRUE means data is processed on a per character basis, rather than on a block basis.

fNoWaitForDialTone

Is a flag. TRUE means the Telephone Service does not wait for a dial tone before dialing.

fReadTimeoutReset

Is a flag. TRUE means the Telephone Service resets its timeout counter for the TsDataRead operation whenever a character is received.

fWriteTimeoutReset

Is a flag. TRUE means the Telephone Service resets its timeout counter for the TsDataWrite operation whenever a character is transmitted.

Section 15
Performance Statistics Service

Overview

The Performance Statistics Service collects measurements of I/O (disk, device, and file system) and processing (processes and partitions). An application can open either a logging session or a statistics session.

During a logging session, the Performance Statistics Service keeps a log of either the current active process in the ready queue or the use of short-lived and long-lived memory.

During a statistics session, the service collects different types of statistics. The types of statistics include processor activity, disk activity, and file system activity, as described below:

- Processor activity statistics include the number of idle process cycles, the number of normal task switches, the number of processes terminated normally and abnormally, and the number of partitions swapped in and out.
- Disk activity statistics include the number of hard disk errors, the number of soft disk errors, the number of IOBs in process, the number of seeks that occurred during read or write of *n* sectors, the number of accesses for a particular extent size during read or write, and the number of logical reads and writes.
- File system activity statistics include the number of files created, renamed, opened, closed, and deleted, the number of files with remade handles, and the number of files with length changes

To use the operations described in this section, first install the Performance Statistics Service on your workstation, as described in the *CTOS Executive Reference Manual* and the *CTOS System Administration Guide*.

Functional Groups of Operations

The following sections offer a brief description of the Performance Statistics Service operations. See the *CTOS Procedural Interface Reference Manual* for a complete description of these operations.

Statistics Session Operations

PsOpenStatSession

Opens a session during which performance statistics are collected by the Performance Statistics Service.

PsGetCounters

Returns the structure containing the performance statistics counters.

PsResetCounters

Resets the performance statistics counters of the given block/index combinations.

Logging Session Operations

PsOpenLogSession

Opens a log for the current active process in the ready queue or use of short-lived and long-lived memory

PsReadLog

Obtains the log for either the process ready queue or the memory usage for the opened logging session.

General-Purpose Operations

PsSCloseSession

Closes the session opened by PsOpenStatSession or by sOpenLogSession.

Statistics Session

Statistics ID Block

A simple statistics ID block is made up of a block number and an index. PsGetCounters and PsResetCounters use this simple statistics ID block. Other operations, such as PsOpenStatSession, require a larger ID block that contains the block number, index, offset, and count of bytes for the requested statistics. (This larger statistics ID block is described in "BlockID," later in this section.)

Block Number

Each general category of statistics corresponds to a given block number, ranging from 1 to 9. For example, the processor statistics are assigned to block number 1. The number of hard disk errors, soft disk errors, and IOBs in process are assigned to block number 2. See Table 15-1, the Performance Statistics Structure, at the end of this section, for the complete listing of the types of counters returned in each of the nine blocks.

Index

The index is a number indicating the position of the disk volume along the workstation bus. The disk closest to the CPU has an index of 0, the next disk has an index of 1, and so on. Statistics for a maximum of 18 disks can be returned for each block. The index must always be specified for blocks 2 through 9 because they collect statistics for a particular disk. The index for block 1 is always 0. (Block 1 collects statistics for the processor.)

Opening a Statistics Session (PsOpenStatSession)

To open a statistics session, use PsOpenStatSession:

PsOpenStatSession (*pbDevName, cbDevName, pbBlockID, cbBlockID, pbShRet*): ercType

where

pbDevName
cbDevName

Are the specification of the device where you want to route the request, in the form *{NodeName}[DeviceName]*

pbBlockID
cbBlockID

Describe a structure that specifies the type of statistics requested and for which devices. (See "BlockID,".)

pbShRet

Describe the memory area of a word variable in which the session handle for this statistics session is returned. (See "Session Handle,".)

Block ID

The statistics ID block used by PsOpenStatSession is an array of records containing the following elements:

Block number	word
Index	word
Offset	word
Count of bytes	word

Use the Performance Statistics Structure (Table 15-1) to determine the appropriate offset and count of bytes. As an example, let's look at block 1 of the Performance Statistics Structure.

Offsets within each block always occur at 4-byte boundaries. For block 1, as shown in Table 15-1, offset 0 specifies idle process cycles, offset 4 specifies normal task switches, and so on. Use the Offset and Count of Bytes array elements to specify where to begin and end statistics reporting within a given block.

For example, the following statistics ID block specifies to count the number of processes terminated normally (offset 8) and abnormally (offset 12):

Block number	1
Index	0
Offset	8
Count of bytes	8

Similarly, this statistics ID block specifies to count the number of partitions swapped in and out:

Block number	1
Index	0
Offset	16
Count of bytes	8

Session Handle

The session handle returned by PsOpenStatSession is used by all subsequent statistics calls (PsGetCounters, PsResetCounters, and PsCloseSession).

Disk Activity

Blocks 4 through 9 collect statistics on certain types of disk activity, as follows. Blocks 4 and 5 count the number of disk seeks for a particular sector size during read or write (that is, the number of 1-sector seeks, number of 2-sector seeks, and so on). Blocks 6 and 7 count how many times a particular extent size is accessed during read or write. The extent sizes increment in powers of 2. Blocks 8 and 9 count the number of logical read or write requests occurring in the system for a particular number of sectors.

For example, suppose you want to request statistics on what extent sizes are accessed most frequently during read and write for the second disk in your system. The statistics ID blocks would be:

Block number	6
Index	1
Offset	0
Count of bytes	512
Block number	7
Index	1
Offset	0
Count of bytes	512

In this case, for example, a greater proportion of accesses may have occurred during both reads and writes to smaller extent sizes. These statistics indicate that the disk is probably fragmented. The user could then squash the disk to improve performance.

Getting the Counters (PsGetCounters)

PsGetCounters instructs the Performance Statistics Service to return counters for specified blocks and indexes:

PsGetCounters: (*sh*, *pbBlockID*, *cbBlockID*, *pbCountRet*, *cbCountRet*): ercType

where

sh

Is the same session handle PsOpenStatSession returns.

pbBlockID
cbBlockID

Specifies the type of statistics requested and for which devices.

pbCountRet
cbCountRet

Describe the memory area the counters must be moved to.

Parameters *pbBlockID* and *cbBlockID* describe the structure that contains the blocks for which you want data. You do not need to request data on all of the blocks specified in PsOpenStatSession. In addition, you can specify the blocks in any order.

For example, suppose you want to collect statistics for two disks in your system on the number of files opened, closed, created, deleted, and renamed, and the number of files with changed lengths. (See block 3 of the Performance Statistics Structure, Table 15-1.) The BlockID arrays specified to PsOpenStatSession would contain the following information:

Block number	3
Index	0
Offset	0
Count of bytes	24
Block number	3
Index	1
Offset	0
Count of bytes	24

The BlockID used with PsGetCounters is a shorter version of the one used to open a session. Only the block number and the index are required. For example, if you specify the following block/index combinations, the counters for the second disk in the system are returned first, followed by the counters for the first disk.

Block number	3
Index	1
Block number	3
Index	0

After calling PsOpenStatSession, your program must stay active until it calls PsGetCounters.

The size of the area the counters return to is the sum of the number of bytes specified at the time of open for each of the block/index combinations. In our example, the area needs to be 48 bytes, since each block/index requires 24 bytes.

Closing a Statistics Session (PsCloseSession)

To close a statistics session, use PsCloseSession (*shRet*).

Logging Session

Logging sessions allow you to keep a record of either the current active process in the ready queue or the use of short-lived and long-lived memory.

Opening a Logging Session (PsOpenLogSession)

You can choose to log either the active processes in the ready queue or to log the use of short- and long-lived memory. When you open the logging session, you allocate a heap for the log in blocks of 512 bytes.

To open a logging session, use PsOpenLogSession:

PsOpenLogSession (*pbDevName*, *cbDevName*, *wBlockID*, *wIterations*, *wLogHeapSize*, *pbShRet*): ercType

where

pbDevName
cbDevName

Are the specification of the device where you want to route the request, in the form *[NodeName][DeviceName]*.

wBlockID

Specifies the type of log, either 10 (for active processes in the ready queue) or 11 (for memory usage).

wIterations

Is the number of iterations you want. (The data returned by PsReadLog indicates the number of *successful* iterations. If the log information does not fit in the given buffer, you may not obtain as many iterations as you asked for.)

wLogHeapSize

Is the size of the log heap, in 512-byte multiples.

pbShRet

Is the memory area where the session handle for this logging session is returned.

For example, to log processor activity, you might specify:

```
PsOpenLogSession(pbDevName, CbDevName, 10, 8, 2048,
pbShRet): ercType
```

Reading a Log (PsReadLog)

To read the log, use PsReadLog:

PsReadLog (*sh*, *pbLogData*, *cbLogData*, *psDataRet*): ercType

where

sh

is the session handle returned by PsOpenLogSession

pbLogData
cbLogData

describe the memory area where the data is to be returned. (See PsReadLog in the *CTOS Procedural Interface Reference Manual* for a description of this structure.)

psDataRet

is the memory address where the actual count of bytes of logging information read is returned. This amount can be less than the total space allotted for the log.

For example, to read the log, you might specify:

```
PsReadLog(Sh, pbLogData, 2048, psDataRet):  ercType
```

In this case, the data returned on each successful iteration is either the ready queue information or the memory usage information. The ready queue information contains the number of processes for that iteration and, for each process, the user number, partition name, and priority. The memory usage information includes the number of partitions for that iteration and, for each partition, the user number, amount of long-lived memory used, and amount of short-lived memory used.

Closing a Logging Session (PsCloseSession)

Use PsCloseSession to close a logging session.

Program Example

Example 15-1 is a sample program that uses the CTOS Performance Statistics Service.

Example 15-1. StatExample.c

```
/*
 * Program title:  StatExample.c
 * Compiler:       MicroSoft C Compiler
 * Description:    This program illustrates the following
 * Performance Statistics System Service calls:
 *   PsOpenStatSession
 *   PsOpenLogSession
 *   PsCloseSession
 *   PsGetCounters
 *   PsResetCounters
 *   PsReadLog
 * This particular Statistics Session collects:
 *   - number of processes terminated normally and
 *     abnormally
 *   - number of hard disk errors, soft disk errors and
 *     IOBs in process for the second disk
 *   - number of files opened, closed, created and
 *     deleted for the second disk
 *   - number of seeks during read for the second disk
 *   - number of reads for the second disk
 * Following is the list of BlockID information required
 * to collect these statistics:
 *      BlockId      Index      Offset      Byte Count
 *      -------      -----      ------      ----------
 *         1           0          8            8
 *         2           1          0           12
 *         3           1          0           16
 *         4           1          0          512
 *         8           1          0          512
 */
#include <string.h>
#include <stdio.h>
```

Example 15-1. StatExample.c (cont.)

```
#define Syslit @
#include <ctosTypes.h>
#define CheckErc @
#define ErrorExit @
#define PSCloseSession @
#define PSGetCounters @
#define PSOpenLogSession @
#define PSOpenStatSession @
#define PSReadLog @
#define PSResetCounters @
#include <ctos.lib.h>

#define MAXSTATS    0x0005
#define MAXDATA     0x0200
char NodeName []    = "{Local}[D0]";
int    sh;
int    rgPsLog [1024];
long    rgPsData [MAXDATA];
struct strId{int number; int index; int offset; int cb;};
struct strCnt {int number; int index;};
struct    strId    rgPsBlockId [MAXSTATS];
struct    strCnt    rgPsBlockCnt [MAXSTATS];
void InitializeFsPsBlocks ();
void OpenFsPsStatsSession ();
void GetFsPsStatsCounters ();
void CloseFsPsStatsSession ()
   {
   /* Use PsResetCounters to reset the associated
    * counters:
    * -    Number of seeks during a read
    * -    Number of reads
    * -    Number of processes terminated normally and
    *        abnormally
    * The BlockId format will contain only the blockId
    * and the index number as follows:
    *
    *          BlockId          Index
    *          -------          -----
    *              4              1
    *              8              1
    *              1              0
    *
```

Example 15-1. StatExample.c (cont.)

```
     *
     * Don't use the block id array as before; just reset.
     */
    CheckErc (PSResetCounters (sh, &rgPsBlockCnt, 12));
    /* Now close the statistics gathering session */
    CheckErc (PSCloseSession (sh));
    }
void GetFileSystemStats ()
    {
    InitializeFsPsBlocks;
    OpenFsPsStatsSession;
    GetFsPsStatsCounters;
    CloseFsPsStatsSession;
    }
void GetProcessorStats ()
    {
    int count;
    /* Open a logging session to get active processes in
     * ready queue
     */
    CheckErc (PSOpenLogSession (&NodeName, sizeof
              (NodeName), 10, 5, 1024, &sh));
    /* Collect the log information */
    CheckErc ( PSReadLog ( sh, &rgPsLog, 1024, &count));
    /* Close the session */
    CheckErc (PSCloseSession (sh));
    }

void GetFsPsStatsCounters ()
    {
    /* Get statistics for the following:
     * -   Number of seeks during a read
     * -   Number of reads
     * -   Number of processes terminated normally and
     *     abnormally
```

Example 15-1. StatExample.c (cont.)

```
     *    The BlockId format will contain only the blockId
     *    and the index number as follows:
     *           BlockId       Index
     *            -------      -----
     *               4           1
     *               8           1
     *               1           0
     */
    /* Set the block id for the number of seeks */
    rgPsBlockCnt[0].number    = 4;
    rgPsBlockCnt[0].index     = 1;
    /* Set the block id for the number of reads */
    rgPsBlockCnt[1].number    = 8;
    rgPsBlockCnt[1].index     = 1;
    /* Set the block id for the number of terminations */
    rgPsBlockCnt[2].number    = 0;
    rgPsBlockCnt[2].index     = 1;
    /* Get the counter values */
    CheckErc (PSGetCounters (sh, &rgPsBlockCnt, 12,
              &rgPsData, MAXDATA));
    }
void InitializeFsPsBlocks ()
    {
    /* Set the block for the number of processes
     * terminated normally and abnormally
     */
    rgPsBlockId[0].number    = 1;
    rgPsBlockId[0].index     = 0;
    rgPsBlockId[0].offset    = 8;
    rgPsBlockId[0].cb        = 8;
    /* Set the block for the number of hard errors,
     * soft errors, and IOBs in process
     */
    rgPsBlockId[1].number    = 2;
    rgPsBlockId[1].index     = 1;
    rgPsBlockId[1].offset    = 0;
    rgPsBlockId[1].cb        = 12;
```

Example 15-1. StatExample.c (cont.)

```
    /* Set the block for the number of files
     * opened, closed, created, and deleted
     */
    rgPsBlockId[2].number    = 3;
    rgPsBlockId[2].index     = 1;
    rgPsBlockId[2].offset    = 0;
    rgPsBlockId[2].cb        = 16;
    /* Set the block for the number of seeks */
    rgPsBlockId[3].number    = 4;
    rgPsBlockId[3].index     = 1;
    rgPsBlockId[3].offset    = 0;
    rgPsBlockId[3].cb        = 512;

    /* Set the block for the number of reads */
    rgPsBlockId[4].number    = 8;
    rgPsBlockId[4].index     = 1;
    rgPsBlockId[4].offset    = 0;
    rgPsBlockId[4].cb        = 512;
    /* Open the Stats Session */
    CheckErc (PSOpenStatSession ( &NodeName,
              sizeof(NodeName), &rgPsBlockId, 40, &sh));
    }
void OpenFsPsStatsSession ()
    {
    /*   Get the statistics for the following:
     *   -   Number of hard errors, soft errors, and IOBs
     *       in process for the second disk
     *   -   Number of files opend, closed, created and
     *       deleted for the second disk
     *
     *   The BlockId format will contain only the blockId
     *   and the index number as follows:
     *
     *      BlockId          Index
     *      -------          -----
     *         2               1
     *         3               1
     */
    /*  Set the block id for the error info */
    rgPsBlockCnt[0].number    = 2;
    rgPsBlockCnt[0].index     = 1;
```

Example 15-1. StatExample.c (cont.)

```
    /* Set the block id for the file usage info */
    rgPsBlockCnt[1].number    = 3;
    rgPsBlockCnt[1].index     = 1;
    /* Get the Ps Counters */
    CheckErc (PSGetCounters (sh, &rgPsBlockCnt, 8,
             &rgPsData, MAXDATA));
    }

void main ()
    {
    /* Get the file system statistics */
    GetFileSystemStats ();
    /* Get the processor statistics */
    GetProcessorStats ();
    /* goodbye */
    ErrorExit (0);
    }

/* Compiled using the following command:
cl
  [Args]  /c /G1 /Gs /Zp1 /AL /I[Sys]<Edf>
          /I[Sys]<MscIncludes>
  [    ]  statexample.c > statexample.lst
```

Example 15-1. StatExample.c (cont.)

```
Linked using the following command:
Link V6
  Object modules              <mscLib>mc0.obj StatExample.obj
  Run file                    StatExample.run
  [Map file]
  [Publics?]
  [Line numbers?]
  [Stack, Dgroup heap size]
  [Max array, data]
  [Min array, data]
  [Run file mode]             pro
  [Version]                   1.0
  [Libraries]                 (msc)
  [DS allocation?]
  [Symbol file]
  [Copyright notice?]
  [File to append]
  [Linker config file]

(MCS) in LinkerConfig.sys is as follows:
:LibraryReference:MSC
:LibraryFile:[sys]<msclib>llibcp.lib
:LibraryFile:[sys]<msclib>libcext.lib
:LibraryFile:[sys]<sys>Ctos.lib
:LibraryFile:[sys]<sys>CtosToolKit.lib
:LibraryFile:[sys]<sys>ENLS.lib
:LibraryFile:[sys]<sys>PsUser.lib */
```

Data Structure

The following data structure is associated with the Performance Statistics Service.

Table 15-1. Performance Statistics Structure

Block Number	Index	Offset	Counts the Number of...
1	0	0	Idle process cycles
	0	4	Normal task switches
	0	8	Processes terminated normally
	0	12	Processes terminated abnormally
	0	16	Partitions swapped into memory
	0	20	Partitions swapped out of memory
2	0	0	Hard disk errors
	0	4	Soft disk errors
	0	8	IOB's in process
	1	0	Hard disk errors
	1	4	Soft disk errors
	1	8	IOB's in process
.	.	.	
.	.	.	
.	.	.	
	17	0	Hard disk errors
	17	4	Soft disk errors
	17	8	IOB's in process
3	0	0	Files opened
	0	4	Files closed
	0	8	Files created

continued

Table 15-1. Performance Statistics Structure (cont.)

Block Number	Index	Offset	Counts the Number of...
	0	12	Files deleted
	0	16	Files renamed
	0	20	File length changes
	0	24	Remade file handles
.	.	.	
.	.	.	
.	.	.	
	17	16	Files renamed
	17	20	File length changes
	17	24	Remade file handles
4	0	0	1-sector seeks during Read
	0	4	2-sector seeks during Read
	0	8	3-sector seeks during Read
.	.	.	
.	.	.	
.	.	.	
	0	504	127-sector seeks during Read
	0	508	128-sector seeks during Read
.	.	.	
.	.	.	
.	.	.	
	17	0	1-sector seeks during Read
	17	4	2-sector seeks during Read
	17	8	3-sector seeks during Read
.	.	.	
.	.	.	
.	.	.	

continued

Table 15-1. Performance Statistics Structure (cont.)

Block Number	Index	Offset)	Counts the Number of
	17	504	127-sector seeks during Read
	17	508	128-sector seeks during Read
5			Same format as block 4 but counts the number of seeks during Write
6	0	0	Accesses of extents 1 sector long during Read
	0	4	Accesses of extents 2 sectors long during Read
	0	8	Accesses of extents 3 sectors long during Read
.	.	.	
.	.	.	
.	.	.	
	0	504	Accesses of extents 127 sectors long during Read
	0	508	Accesses of extents 128 sectors long during Read
.	.	.	
.	.	.	
.	.	.	
	17	0	Accesses of extents 1 sector long during Read
	17	4	Accesses of extents 2 sectors long during Read
	17	8	Accesses of extents 3 sectors long during Read

continued

Table 15-1. Performance Statistics Structure (cont.)

Block Number	Index	Offset)	Counts the Number of
.	.	.	
.	.	.	
.	.	.	
	17	504	Accesses of extents 127 sectors long during Read
	17	508	Accesses of extents 128 sectors long during Read
7			Same format as block 6 but gives the number of accesses during Write
8*	0	0	1-sector logical Reads
	0	4	2-sector logical Reads
	0	8	3-sector logical Reads
.			
.			
.			
	0	504	127-sector logical Reads
	0	508	128-sector logical Reads
.			
.			
.			
	17	0	1-sector logical Reads
	17	4	2-sector logical Reads
	17	8	3-sector logical Reads
9*			Same format as block 8 but it gives the number of logical Writes

*The appropriate counter is incremented with each Read or Write operation.

Section 16
Asynchronous System Service Model

Introduction to Asynchronous Services

This section describes the asynchronous system service model. The asynchronous service can be thought of as a thread service. It allows the user to create multiple threads of execution within a single process. It provides greater throughput than a single-threaded system service.

This section provides you with an overview of the asynchronous model and instructions on how to write a program using library procedures contained in *Async.lib*. *Async.lib* is available on the CTOS Development Utilities diskettes. The following operating systems support these procedures:

Use of the asynchronous system service model is transparent to applications running on these operating systems.

Note: *Because the procedures contained in* Async.lib *use global variables, these procedures are not designed for multithreaded programs.*

Terminology

Key terms used throughout this section are described below.

A request-based system service uses interprocess communication (IPC) to provide services to client programs. Although this section provides some review, this section assumes your knowledge of IPC, system services, and request processing. For additional information, refer to the sections on these subjects in the *CTOS Operating System Concepts Manual*.

With synchronous processing, one transaction at a time is completed before another is begun. Synchronous processing is probably familar to you if you've ever had to wait your turn in a supermarket checkout line. Perhaps you have experienced a situation similar to the following.

Although you are next in line with your single pint of ice cream in hand, you have to wait because the customer at the counter is writing a check. Fortunately, the customer doesn't take too much time, so you think the wait will be short. Then the clerk must call the manager to approve the check, but the manager is busy with another customer. Minutes drag by while the clerk idly jingles change in the cash drawer. You wish the clerk would use this idle time to handle your purchase, but the clerk can process only one transaction at a time.

Many request-based system services are designed along the lines of the supermarket checkout system. A system service that operates synchronously cannot start processing the next client queued until the system service has completed the entire transaction for the current client: if the system service must send out a request of its own on behalf of the current client, nothing can be done for the next client queued. The system service simply waits at its default response exchange for the response. This type of request-based system service is called a synchronous model of system service.

With asynchronous processing, more than one transaction can be handled at a time. The asynchronous model of system service is designed to use asynchronous processing. Instead of waiting idly for a response to a request it sent out, an asynchronous system service can process a new request or a response that arrives at its service exchange. As such, the asynchronous system service model provides an attractive alternative to traditional synchronous processing.

Contexts are the key design element of the asynchronous processing. A context is an individual thread of execution that has its own stack history (such as local variables). An asynchronous system service process consists of multiple contexts, each sharing the system service process stack.

Stack sharing is possible because each context has a unique stack pointer. Upon execution of a context, the stack pointer is set to point to the stack for that context. Before a second context executes, the stack of the first context is saved in a memory structure. Then the stack pointer is set to the appropriate location for executing the second context. Because the stack of a context can be saved, each context can be resumed where it stopped executing simply by setting the stack pointer to the appropriate location. As described later in this section, contexts can be easily created, saved, resumed, or terminated using context procedures in *Async.lib*.

A heap is used for the dynamic allocation and deallocation of memory in context handling. The heap used by the asynchronous library procedures is a linked list of free memory blocks in the data segment (DS) space.

Synchronous and Asynchronous System Service Models

The asynchronous model has several advantages over the synchronous one.

For each model, the main process is a Wait loop. Although the models are similar in this respect, they have basic underlying differences.

First, let's examine the Wait loop for the synchronous model. A simplified version of the loop code is shown below:

```
while (1)
{
   erc = Wait(exchServ, &pRq);
   erc = ProcessRequest();
   pRq->ercRet = erc;
   erc = Respond(pRq);
}
```

In this model, the system service waits for requests at its service exchange. When a request arrives, the system service processes the request and responds to the client. Then the system service returns to the top of its Wait loop for the next request to process.

Figure 16-1 illustrates the program flow for this model. (The decision box following the box labeled "Process request" shows what transpires in the course of processing a request.) In some cases, a system service can process a request and respond to the client immediately without making additional requests to external agents on behalf of the client. If, however, the system service needs to request the services of an external agent, the system service must wait at its default response exchange for the response. In a simple case, such as when a Read request is sent to the operating system, the wait is relatively short. On the other hand, if a long document is sent to a device driver for printing, the wait is a significant period of time. In either case, the system service cannot do any other useful work until the response arrives.

Now, let's examine the asynchronous model. A simplified version of the Wait loop code is shown below:

```
while (1)
{
   erc = Wait(exchServ, &pRq);
   if (pRq->exchResp == exchServ)
   {
       erc = ResumeContext();
   }
   else
   {
       erc = ProcessRequest();
       pRq->ercRet = erc;
       erc = Respond(pRq);
   }
}
```

Figure 16-1. Program Flow for the Synchronous Model

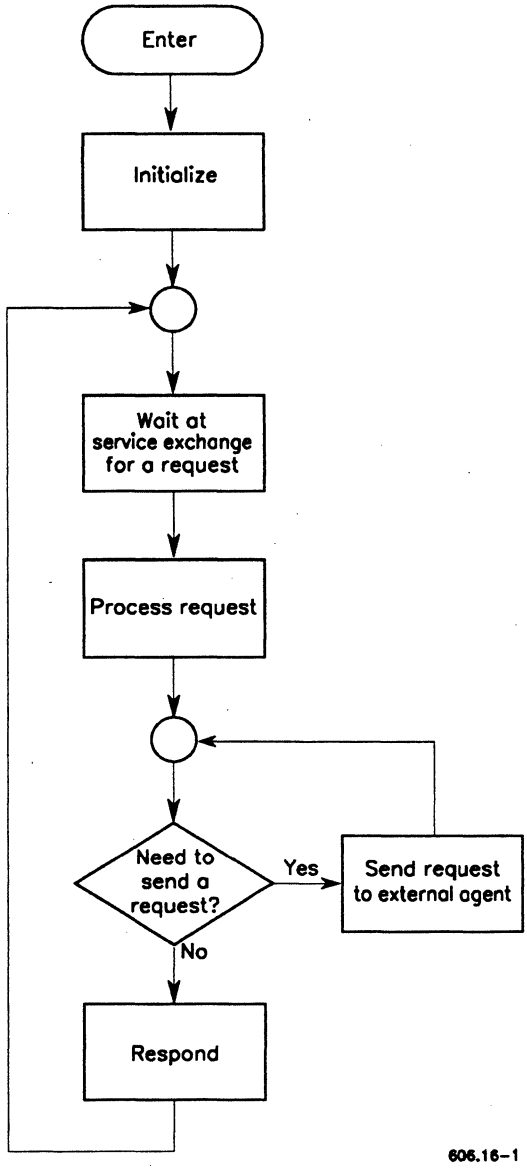


Figure 16-2 illustrates the program flow for this model. The system service waits for either a request or a response. If a request arrives, the system service can process that request. If the system service needs to send a request to an external agent, it does not wait for the response. Instead, it sends the request (using one of the asynchronous request formats described later in this section). The context that sent the request is saved, and the system service process returns to the top of its Wait loop to wait for other requests or responses to arrive.

If a response from an external agent arrives (the response exchange in the request block is the system service exchange), the context that originally sent the request is resumed. In Figure 16-2, this program flow is represented by the arrow pointing from the box labeled "Resume context" to the box labeled "Process request." (Request processing actually resumes at the program statement immediately following the asynchronous request to the external agent.)

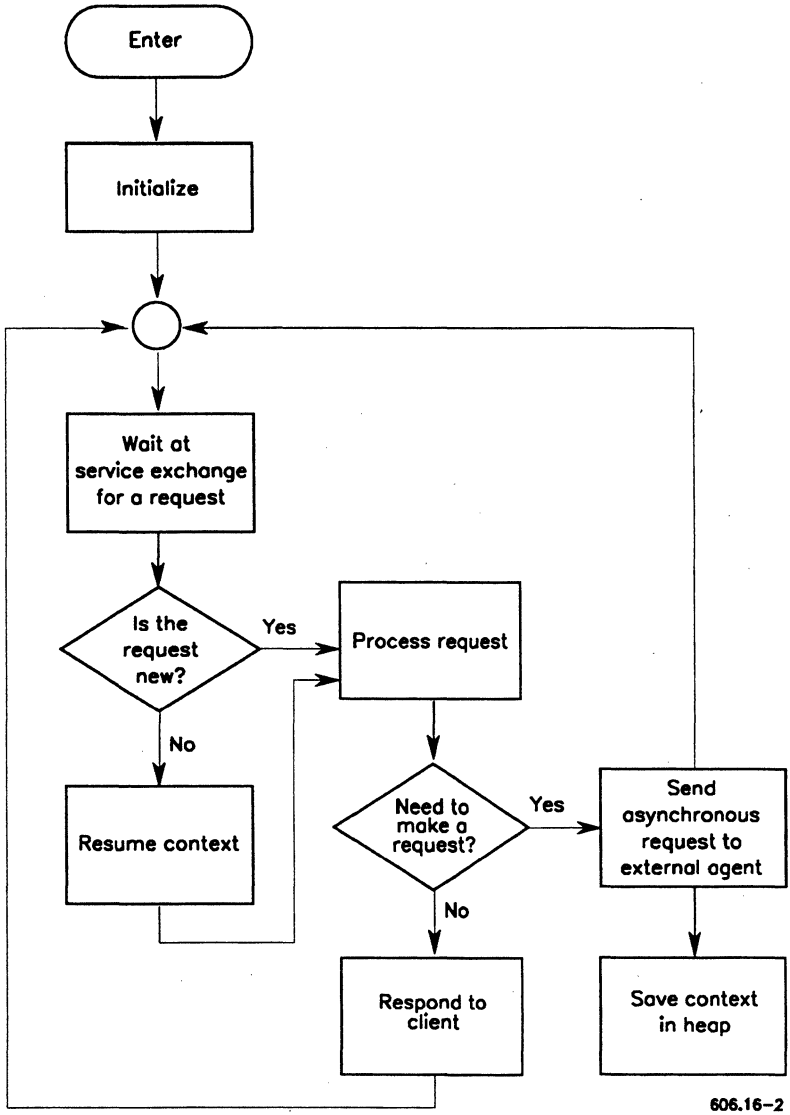
Because of the continuous flow of activity among executing contexts, the asynchronous model can handle several clients within the timeframe required for the synchronous model to handle a single client.

Writing an Asynchronous System Service

The code that is common to all services you may write in an asynchronous system service is provided in a single C source module. (See the module *AsyncService.c* in "Program Example," at the end of this section.) This module is called the common-code module. The common-code module performs the following functions:

- Initializes the system service
- Waits for incoming requests or responses (Wait loop)
- Manages (creates, saves, resumes, and terminates) contexts
- Responds to a deinstallation request

Figure 16-2. Program Flow for the Asynchronous Model



Note: *Using the asynchronous model does not preclude using synchronous requests or the standard procedural interface. You can use synchronous requests in some cases.*

If you have written synchronous system services, you should be familiar with the installation and deinstallation procedures illustrated in this module. (Refer to "System Services Management," in the *CTOS Operating System Concepts Manual.*) As previously mentioned, the Wait loop and context managment are unique to the asynchronous model.

In addition to performing the functions just described, the common-code module refers to external variables you need to declare and procedures you need to write in a module containing the code that is unique to your asynchronous system service. Because the module you must write will contain the main program for your particular system service, this module is called the main module. Your main module serves the requests defined for your service. It also contains any additional code necessary for initialization.

To create the run file for your system service, you compile your main program source code and link the resulting object module with the common-code module. Figure 16-3 illustrates this procedure.

More information on linking your program is contained in "Linking Your System Service," later in this section. For now, you should keep in mind that you need to write only part of the source code for your system service run file.

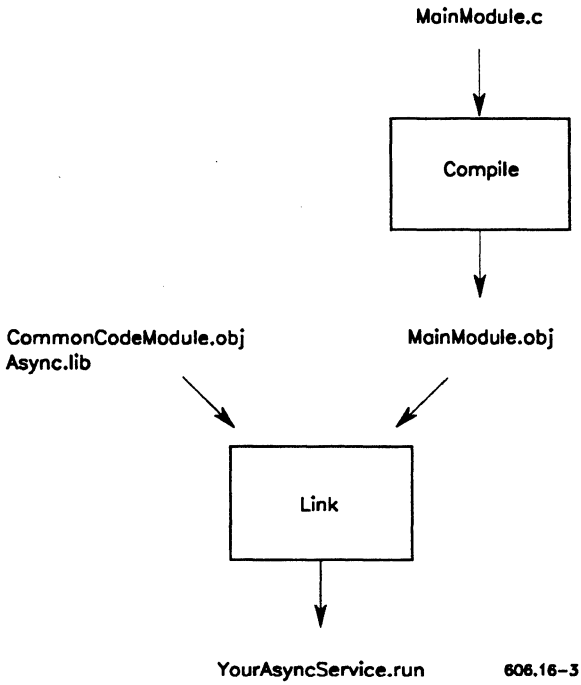
In "Program Example," at the end of this section, you also will see the source to four programs: *Example.c*, *Deinstall.c*, *Start.c*, and *Stop.c*. These programs are an example of a user-written system service. (See your release documentation for details on all the files you need to run this example program.)

Example.c is the main module. This program filters file system requests and writes to a recording file the names and access times of all open files. To write your main module source code, you can use Example.c as a template.

Start.c, *Stop.c*, and *Deinstall.c* support the filter service. These programs are compiled separately and are used by a command interpreter such as the Executive for activating and deactivating the filter service's recording process and for deinstalling of the filter.

For further information on these programs, see "Program Example."

Figure 16-3. Source Modules to Run File



Async.lib Procedures

All the *Async.lib* procedures are described next in this section. The procedures are presented in two categories: those used in your main module and those used by the common-code module. To see how the procedures are used in an actual application, refer to the example programs at the end of this section.

Async.lib Procedures You Can Use in the Main Module

Requesting a Service on Behalf of a Client

The easiest way to make a request on behalf of a client is to use the asynchronous request procedural interface. Because the asynchronous request procedural interface is similar to its synchronous counterpart, let's review the synchronous request procedural interface.

Review of the Synchronous Request Procedural Interface

If you are familiar with the synchronous procedural interface, you realize that its strength lies in its ease of use: to invoke the synchronous procedural interface, you simply write a statement of the following form:

erc = RequestName(*arg0, arg1,...,argn-1*);

The synchronous procedural interface, guided by a set of operating system tables addressed by the request name, builds a request block on the caller's stack using the parameters passed by the caller. To send the request to the operating system, the procedural interface calls the kernel primitive Request. The operating system, in turn, redirects the request to the appropriate service exchange. (For details on Request, see the *CTOS Procedural Interface Reference Manual.*)

After sending the request, the synchronous procedural interface waits for a response at the default response exchange. When the response arrives, the procedural interface removes the request block from the stack, and the calling program continues with the next statement to be executed.

Asynchronous Request Procedural Interface

You can access the asynchronous request procedural interface by using either of two procedures in *Async.lib*: BuildAsyncRequest or BuildAsyncRequestDirect.

BuildAsyncRequest. This procedure sends a request to the exchange serving the request. To invoke BuildAsyncRequest, you use the same request parameter list you would provide to the synchronous procedural interface except that you add the request code to the end of the list. For example, in C language, your procedure call would appear as follows:

erc = BuildAsyncRequest(*arg0, arg1,...,argn-1, rcRequestCode*);

where

rcRequestCode

Is the request code of the specified request.

arg0, arg1,...,argn-1

Are the parameters for the specified request. (For details on the parameters for all CTOS requests, see the *CTOS Procedural Interface Reference Manual.*)

If you are using PL/M, passing a variable number of parameters is somewhat different. For details, see "Passing a Variable Length Parameter List in PL/M," later in this section.

Like the synchronous request procedural interface, BuildAsyncRequest uses tables in the operating system to build the request block on the stack using the arguments passed by the caller. When the request block is built, BuildAsyncRequest calls Request to send the request to the operating system.

Here, however, the similarity between the synchronous and asynchronous procedural interface ends. Instead of waiting at the system service's default response exchange for the request to come back, the asynchronous procedural interface saves the state of the system service context currently running. (Code in the common module takes care of handling contexts. For details on the context management procedures, see "Managing Contexts," later in this section.) The asynchronous procedural interface then returns to the top of the system service Wait loop to wait for the next request or a response.

BuildAsyncRequestDirect. This procedure sends a request to a specified exchange. The parameters to BuildAsyncRequestDirect are the same as those to BuildAsyncRequest except that BuildAsyncRequestDirect takes the target exchange as an additional parameter. For example, to invoke BuildAsyncRequestDirect in C language, code your procedure call as follows:

erc = BuildAsyncRequestDirect(*arg0, arg1,...,argn-1, rcRequestCode, exchTarget*);

where

rcRequestCode

Is the request code number of the specified request.

exchTarget

Is the number of the target exchange.

Passing a Variable Length Parameter List in PL/M

To call a procedure and pass a variable number of parameters in PL/M, your program needs to make an indirect procedure call through a pointer variable, as shown below:

```
DECLARE procBuildAsyncRequest POINTER EXTERNAL;
DECLARE ercAsync ErcType EXTERNAL;

CALL procBuildAsyncRequest(arg0, arg1,...,argn-1,
rcRequestName);
erc = ercAsync;
```

Two problems arise when making indirect procedure calls in PL/M.

First, a procedure called indirectly may not return a value. For this reason, the status code (erc) is returned in the global variable ercAsync. (The variable ercAsync is declared publicly in the common-code module.)

Second, the compiler does not perform any type conversion of parameters. This problem would occur, for example, if there were literals in the parameter list because the compiler would not know how to convert them. Consider the following example of a call to OpenFile using the synchronous procedural interface:

```
erc = OpenFile(@fh, @rgbFileName, cbFileName, 0, 0,
modeRead);
```

The first 0 parameter represents a pointer value (pbPassword), and the second, a word value (*cbPassword*). The compiler knows this because OpenFile has been explicitly declared. It appears to follow that to use the asynchronous procedural interface, you might (erroneously) code a call as shown below:

```
CALL procBuildAsyncRequest(@fh, @rgbFileName, cbFileName,
    0, 0, modeRead, rcOpenFile);
erc = ercAsync;
```

However, this example is incorrect. Instead of converting the 0 values, the compiler would push a byte value onto the stack for each value. This would not work in PL/M or in C. Ways to circumvent this problem are described next.

Word values. The easiest way to call explicitly for a word value in PL/M is to use the INT function. INT(0) causes a word value to be pushed onto the stack. The request code (last parameter) passed to BuildAsyncRequest must be a word value. An example of how to declare the INT function is shown below:

```
DECLARE rcOpenFile LITERALLY 'INT(4)';
```

Pointer values. The easiest way to generate a 0 pointer value in PL/M is to declare a pointer variable globally with a value of 0, for example,

```
DECLARE pZero POINTER INITIAL(0);

CALL procBuildAsyncRequest(@fh, @rgbFileName, cbFileName,
pZero, INT(0), modeRead, rcOpenFile);
erc = ercAsync;
```

Caution

It is extremely important that you check the validity of the variable length parameter list. If the number and type of arguments passed are not correct, results of program execution are unpredictable.

Building Request Blocks

Rather than having the asynchronous procedural interface automatically build the request block, you can build the request block yourself to improve performance. When building a request block, your system service can use either the AsyncRequest or the AsyncRequestDirect procedure to send the request block to the operating system.

AsyncRequest

This procedure uses the Kernel primitive Request to send the request block to the operating system for routing to the appropriate service exchange. AsyncRequest then returns to the top of the system service Wait loop. When the response arrives, the context is resumed at the next program statement.

To use AsyncRequest, code your procedure call as follows:

erc = AsyncRequest (*pRq*);

where

pRq

Is the memory address of the request block.

AsyncRequestDirect

This procedure uses the Kernel primitive RequestDirect rather than Request to send the request block to a specified exchange. (For details on RequestDirect, see the *CTOS Procedural Interface Reference Manual.*) AsyncRequestDirect then returns to the top of the system service Wait loop. When the response arrives, the context is resumed at the next program statement.

To use AsyncRequestDirect, code your procedure call as follows:

erc = AsyncRequestDirect (*exch*, *pRq*);

where

exch

Is the specified system service exchange.

pRq

Is the memory address of the request block.

Checking the Context Stack

All the asynchronous request procedures (BuildAsyncRequest, BuildAsyncRequestDirect, AsyncRequest, and AsyncRequestDirect) ensure that the context stack has not overflowed. If you write a procedure that does not call any of these request procedures, a stack overflow could go undetected. To validate the context stack in such a case, your program can call the procedure CheckContextStack.

CheckContextStack

To use CheckContextStack, code your procedure call as follows:

erc = CheckContextStack();

Heap

The heap is a linked list of free memory blocks in the DS space. Initially, the list contains only one large block of memory. When a request for heap space is made, the free list is searched until a block large enough for the memory requirements of the request is found. If the block is exactly the size requested, the block is removed from the free list and returned to the caller. If the block is larger than the amount of memory requested, however, the block is split into two parts: one is returned to the caller and the other to the free list. The free list is maintained in order of increasing addresses, allowing blocks that are returned to be merged with adjacent blocks in the list.

The common-code module initializes the heap. Although the common-code module uses the heap management procedures to manage contexts, you can use these procedures to allocate and deallocate memory in your main module as well.

Allocating and Deallocating Heap Memory

Your main module can use the heap procedures HeapAlloc and HeapFree to allocate and free memory blocks, respectively, during program execution. Rather than storing all variables in the stack, it is more efficient for your program to use these procedures to allocate storage for large blocks of data as the storage space is needed.

HeapAlloc. HeapAlloc allocates memory from the heap. To use HeapAlloc, code your procedure call as follows:

erc = HeapAlloc(*cBytes*, *poMemoryRet*);

where

cBytes

Is the count of bytes to allocate from the heap.

poMemoryRet

Is the memory address into which the pointer to the allocated memory is returned.

HeapAlloc returns status code 4533 (No heap memory available) if the heap does not have not enough contiguous memory to meet the requirements of the request.

HeapFree. This procedure returns memory to the heap. To use HeapFree, code your procedure call as follows:

erc = HeapFree(*pMemory*);

where

pMemory

Is the pointer to the block that was previously allocated from the heap.

HeapFree returns status code 4534 ("Invalid heap block") if the pointer passed to HeapFree is not the address of a valid heap block.

Conserving Heap Memory

Your program can run out of heap memory if care is not taken to conserve it. Status code 4533 (Heap memory not available) is returned when this happens. To avoid running out of memory, your program should allocate large blocks for a procedure's data storage when a procedure begins. The memory should be returned to the heap as soon as the procedure is finished. You also should ensure that none of your procedures uses an excessive amount of stack space for local variables. A single procedure of this nature can inflate the stack size, wasting memory.

System Requests

You need to define system requests for abort, termination, and swapping as you would for any system service you write. A general discussion of the significance of these requests follows. (For additional information, see the *CTOS Operating System Concepts Manual.*)

Terminating and Aborting Requests

The operating system issues a termination or abort to guarantee that

- No requests are returned to the program after it has been terminated
- No outstanding requests will access memory associated with a program that has been terminated
- Connection-oriented services in use by the program are closed (for example, file handles)

When any system service (synchronous or asynchronous model) receives a termination or abort request, the service is required to respond to all requests with the same user number and then to respond to the abort/termination request.

If an asynchronous system service does not have any active context(s) for the terminating user number, the system service can simply respond to the abort/termination request.

If, on the other hand, the asynchronous system service does have active context(s) for the terminating user number, the service has one outstanding client request for each context. In addition, one outstanding asynchronous request is issued on behalf of the client for each context. In this situation, the asynchronous system service should respond to the client request as soon as possible so that it can respond to the abort/termination request.

Swapping Requests

In versions of the CTOS operating system prior to CTOS III, the operating system issues swapping requests for reasons similar to abort/termination requests, namely to guarantee that

- No requests will be returned to a program after it has been swapped out of memory
- No outstanding requests will access memory associated with a program that is swapped out

When any system service (synchronous or asynchronous model) receives a swapping request, the system service is required to respond to all requests with the same user number and then to respond to the swapping request.

When handling swapping requests in your asynchronous program module, you should consider the following: when control returns to a context after the return of an asynchronous request, the client request being handled may not be the same client request the context started with. The request could have been reissued by the operating system when the client program was swapped back into memory. In this case, the pointers in the client request block may not be the same as they were the first time the request was received. If the request came over the cluster, the request block most likely was received in a different buffer, and the pointers were constructed to point to data in that buffer. For this reason, a system service must not save away any pointers from a client request between asynchronous requests. The following sequence of events illustrates this point:

1. Client A's request, OpenYourFile, arrives at the system service exchange.
2. The system service saves (in variable *pbPassword*) the memory address of the password located in the OpenYourFile request block. This causes *pbPassword* to be saved on the stack as part of the system service's current context (Context 1).
3. The system service makes an asynchronous request to read a file on behalf of Client A.
4. Context 1 is saved, and the system service returns to the top of its Wait loop to perform other work.
5. While the system service is performing other work, Client A is swapped to disk.
6. The response to the asynchronous read request arrives, and Context 1 is resumed.
7. The system service attempts to use the password pointed to by *pbPassword* to open a file.

It is possible that a status code will be returned when the system service attempts to open the file using the address at pbPassword. When Client A was swapped back into memory, the operating system issued a new OpenYourFile request. Upon reissuing the request, the operating system may have placed the request at a different location. If so, the request block pointer saved in *pbPassword* is no longer valid.

Handling System Requests

Because handling system requests is often the most difficult and bug-prone part of writing a system service, two *Async.lib* procedures are provided: TerminateContextUser and SwapContextUser.

Handling Termination and Abort Requests

To handle termination and abort requests, your program can use the TerminateContextUser procedure.

TerminateContextUser. This procedure responds to all client requests for a given user number. Transparent to the caller, it waits for all outstanding asynchronous system service requests to return.

To use TerminateContextUser, code your procedure call as follows:

erc = TerminateContextUser(*userNum*, *erc*);

where

userNum

Is the user number associated with the terminating program. The user number is obtained from the termination request.

erc

Is the status code number.

TerminateContextUser uses the following algorithm:

1. If there are no active context(s) for the terminating user number, control is returned to the caller.
2. If there are outstanding context(s), TerminateContextUser sets a terminated flag for each context to be terminated and responds to the client request immediately if possible (if all the pb/cb pairs point to system service data).
3. If TerminateContextUser can respond to all the outstanding client requests immediately, control is returned to the caller. If TerminateContextUser must wait for one or more asynchronous requests to finish, it returns to the system service Wait loop rather than to the next instruction to execute (following the call to TerminateContextUser). This action suspends the context that called TerminateContextUser (that is, the client of the abort/termination request).
4. Each time an asynchronous request returns, its context is resumed by the procedure ResumeContext. (Resuming contexts is handled by the common-code module and is described in "Managing Contexts," later in this section.) The ResumeContext procedure, however, does not resume a context flagged as terminated. Instead, ResumeContext calls TerminateContext (also described in "Managing Contexts"). If the client request is still outstanding, TerminateContext responds to the request. If the request is the last outstanding client request, TerminateContext resumes the context that called it.

Handling Swapping Requests

To handle swapping requests, your program can use the SwapContextUser procedure.

SwapContextUser. To use this procedure, code your procedure call as follows:

```
erc = SwapContextUser(userNum);
```

where

userNum

Is the user number associated with the terminating program.

SwapContextUser works very much like TerminateContextUser. With SwapContextUser, however, the system service is not finished with the client request. When the user is swapped back into memory, the context must resume execution where it left off.

The following steps describe how the context procedures handle swapping:

1. SwapContextUser responds to each client request with status code 37 ("Service not completed"), causing the operating system to reissue the request when the program is swapped back into memory. This action is transparent to the application program that originally issued the request.
2. When a new request arrives, the system service calls CreateContext. (CreateContext is called in the common-code module. For details, see "Managing Contexts.") CreateContext checks to see whether the user number field of the swapping request has an associated context waiting to be swapped back into memory.
3. If CreateContext finds an associated context, the request was one reissued by the operating system (see step 1). In this case, CreateContext searches for the context associated with the same user number and request code as the request that just arrived. Upon finding the context, CreateContext resumes it rather than creating a new context.

Debugging Aids

The Asynchronous Services provide a method of logging for use in debugging efforts.

Examining the Logging Module LogAsync

Async.lib contains a module called LogAsync that you can link with your program for debugging purposes. (If you do not link with LogAsync, however, you must link with the module LogAsyncDmy to resolve references. For details, see "Linking Your Program," later in this section.)

LogAsync contains messages logged by the common-code module when that module calls the following procedures:

- LogMsgIn
- LogRequest
- LogRespond

(For details on these procedures, see "Logging Messages for Debugging," later in this section.)

LogAsync contains a trace buffer (rgLog), which is an array of 50 nine-word entries. Each entry has the format shown below:

Offset	Field	Size (bytes)	Description
0	*code*	2	One of four hexadecimal values (described below)
2	*pRq*	4	Address of the message
6	*rq*	12	Request block header

The hexadecimal value of the field *code* can be any of the following:

Value	Description
AAAA	A message is received at the system service exchange. This entry is made when LogMsgIn is called.
BBBB	A request was sent by BuildAsyncRequest or BuildAsyncRequestDirect. This entry is made when LogRequest is called.
CCCC	A response to an asynchronous request was received at the system service exchange. This entry is made when LogMsgIn is called.
FFFF	A client request was responded to. This entry is made when LogRespond is called.

The rgLog array is a ring buffer that starts filling from the last buffer entry and fills toward the top of the buffer. This arrangement makes the buffer more convenient to examine with the Debugger. (For details on using the Debugger, see the *Debugger User's Guide*.)

The pointer variable *pLog* points to the most recent buffer entry in rgLog. To examine rgLog in the Debugger,

Type **pLog** (Press **CODE+RIGHT ARROW**.)

The Debugger displays the word at pLog.

To see the entire contents of most recent entry,

Press **DOWN ARROW** eight times.

By pressing **DOWN ARROW** nine more times, the Debugger displays the contents of the previous entry, and so forth.

Maintaining Debugging Statistics

The *Async.lib* procedures maintain statistics that are helpful when debugging and tuning system services that use the library procedures. You can examine the global label AsyncStats while in the Debugger or when looking at a dump. This label is followed by the statistics described below.

nStackOverflow

Is the number of times stack overflow was detected by BuildAsyncRequest or BuildAsyncRequestDirect.

nCreateContextError

Is the number of times CreateContext returned a status code because there was no heap space available to create a context.

nResumeContextError

Is the number of times ResumeContext returned a status code because the global variable *pRq* did not point to a request block that belonged to an active context. (For details, see "ResumeContext," later in this section.)

minStackFree

Is the minimum number of bytes free on the stack after building a request block. This value takes into account the space required for all request overhead plus the 64 bytes required to handle an interrupt.

minHeapFree

Is the minimum number of bytes free in the heap.

nBytesHeap

Is the number of bytes allocated to the heap.

nBytesHeapFree

Is the number of bytes currently free in the heap.

nCcbActive

Is the number of currently active contexts.

nCcbActiveMax

Is the maximum number of contexts that were active at any one time.

Async.lib Procedures Used by the Common-Code Module

The procedures described next are used in the common-code module.

Managing Contexts

The common-code module uses *Async.lib* procedures to manage contexts. A context is an instance of program execution along with all the program's local variables. Local variables are variables allocated on the stack. In this sense, a context is very much like a CTOS process. (For details on CTOS processes, see "Process Management" in the *CTOS Operating System Concepts Manual.*) The difference is that, with the asynchronous system service model, several contexts, or threads of execution, can exist at any given time within a single CTOS process. These contexts may be easily created, saved, restored, and terminated.

As mentioned earlier in this section, context handling is a key concept of asynchronous processing. When the system service receives a request, it allocates stack space from the heap for a structure called the Context Control Block (CCB). If the system service needs to send an asynchronous request to an external agent, the state of the currently executing context is saved in the CCB. This frees the system service to return to its Wait loop for processing other incoming requests or responses. When the response to the asynchronous request arrives, the context is resumed, allowing the system service to continue processing the client's request where it left off at the time it sent the request. When processing of the client's request is complete, the system service terminates the context and returns any memory allocated for it back to the heap.

Although contexts typically are associated with request blocks, this is not always the case. Contexts are created whenever any type of message is received. (For details, see "Other Ways Contexts Can Be Used," later in this section.)

To manage contexts, the common-code module uses the following *Async.lib* procedures:

- CreateContext
- ResumeContext
- TerminateContext

Creating a Context

CreateContext. This procedure allocates stack space for a context from the heap. To conserve heap memory, stack size for a context is limited to approximately 300 bytes.

The procedure call to CreateContext is as follows:

erc = CreateContext(*stackSize*, *userNum*);

where

stackSize

Is the number of bytes to allocate for the stack of the context to be created.

userNum

Is the user number associated with the context.

The user number is used by the abort/termination procedures described in "Handling System Requests," earlier in this section. If the context is associated with a timer request block (TRB) rather than a client request, the user number should be 0. (For details on TRBs, see "Other Ways Contexts Can Be Used," later in this section.)

When control returns from CreateContext with erc = ercOK, the stack pointer is changed. The caller, therefore, must not use any local variables upon the return because they are no longer addressable.

If no memory is available from the heap, status code 4533 ("Heap memory not available") is returned. Your main module also can allocate heap memory during program execution. For ways to conserve memory, see "Heap," earlier in this section.

Context Control Block

CreateContext allocates a CCB from the heap and sets a global offset to point to it. The CCB has two parts: the header and the stack.

The header of the CCB is accessed only by the asynchronous system service library procedures. It contains the following information:

- Pointers to a doubly linked list of active CCBs.
- A list of heap blocks allocated to the context. When the context is terminated, any heap blocks associated with the context are freed.
- A global pointer (pRq) to the client request being served by this context. CreateContext saves the request block pointer in the CCB header so that the pointer can be restored when the context is resumed.
- The total size of the CCB.
- A seal to ensure that the CCB is not overwritten.

The stack portion of the CCB starts at the end of the CCB and grows toward the header.

Resuming a Context

ResumeContext. This procedure resumes execution of a context at the point where execution left off the last time the system service made an asynchronous request to an external agent. If a request was built on the stack, the request block is removed from the stack, and the status code returned in the request block is stored in the global variable ercAsync.

The procedure call to ResumeContext is as follows:

```
erc = ResumeContext();
```

ResumeContext does not take any arguments. Instead, it uses the pointer to the last request block received (stored in the global variable pRq) to locate the context to resume. Control does not return from ResumeContext unless an error is detected. If pRq does not point to a request block sent out by either BuildAsyncRequest or BuildAsyncRequestDirect, status code 4532 ("Context not found") is returned.

Terminating a Context

TerminateContext. This procedure is used to terminate a context and return its stack space to the heap. The procedure call to TerminateContext is as follows:

```
erc = TerminateContext();
```

Like ResumeContext, TerminateContext does not take any arguments. It uses a global offset that points to the current CCB in the heap. Any heap space associated with the context is returned to the heap at this time. After terminating the context and returning the heap memory, TerminateContext returns to the caller's Wait loop. A status code is returned only if the offset to the current CCB does not point to a valid context.

Other Ways Contexts Can Be Used

The common-code module (*AsyncService.c* shown at the end of this section) illustrates how the procedure HandleRequest creates a context each time a request is received.

Although a context typically is associated with a request that is being processed, this is not always the case. A context also can be created to service a timeout when a TRB is received. In the common-code module, the main process procedure WaitLoop can create a context whenever it receives any kind of message. The message does not need to be a request, nor does the pointer to the message need to be unique. For example, the timeout procedure could send *N* messages to the system service exchange to cause *N* contexts to be created. When a context is created, the value of the pointer to the last message received (represented by the global variable *pRq*) is saved in the new CCB, and the value is restored whenever a context is resumed.

The example program at the end of this section does not show how to handle timers, although there is provision for calling a HandleTimer function in the common-code module (*AsyncService.c*). The main program (*Example.c*) shows how to call HandleTimer. (See *AsyncService.c*. Note the call to HandleTimer in the HandleRequest function. Then examine the corresponding HandleTimer function in *Example.c*.)

Terminating Contexts at Deinstallation

The system service receives a request when it is time to deinstall. The steps for deinstalling an asynchronous system service are the same as those for deinstalling a synchronous service. (For an enumeration of these steps, see the *CTOS Operating System Concepts Manual.*) However, the common-code module of an asynchronous system service must perform one unique procedure to terminate each active context.

To perform this procedure, the common-code module calls TerminateAllOtherContexts. The call to TerminateAllOtherContexts is as follows:

```
erc = TerminateAllOtherContexts(erc);
```

where

erc

Is the status code that is responded to for any outstanding requests associated with the context being terminated.

TerminateAllOtherContexts terminates all contexts except for the calling context. The effect of this procedure is to call TerminateContextUser (described in "System Requests," earlier in this section) for every active user, including the system service. This causes any contexts associated with the system service (such as a context handling a TRB) and client contexts to be terminated.

Using the Heap

The common-code module uses a heap to manage contexts. When a context is created, memory for it is allocated from the heap. The memory is returned to the heap when the context is terminated.

The heap is allocated out of DS space. Using the asynchronous system service library procedures, DS and SS are equivalent. The advantages of this arrangement are

- All pointers can be near pointers. (All memory addresses can be offsets within the same segment.)
- Code size is smaller and faster.
- Debugging is easier.

Note: *The combined memory of the heap, static data, and the main program stack cannot exceed 64K bytes.*

Managing the Heap

To manage the heap, the common-code module uses the following heap management procedures:

- HeapInit
- HeapAlloc
- HeapFree

As described in "Heap," earlier in this section, the HeapAlloc and HeapFree procedures also can be used during program execution of your main program module.

HeapInit. This procedure initializes the heap. HeapInit is called before any of the asynchronous system service library procedures are used. The heap is initialized with a fixed amount of memory. Once this memory is allocated, it cannot be deallocated or changed in size. Because a system service cannot allocate any more memory once ConvertToSys is called, this is not a disadvantage.

The procedure call to HeapInit is as follows:

erc = HeapInit(*cBytes*, *pHeap*);

where

cBytes

Is the count of bytes to be used for the heap.

pHeap

Is the pointer to the first byte of heap space.

The heap space is usually memory allocated by AllocMemoryInit. (For details, see "AllocMemoryInit," later in this section.)

HeapAlloc. This procedure allocates memory from the heap. The procedure call to HeapAlloc is as follows:

erc = HeapAlloc(*cBytes*, *ppMemoryRet*);

where

cBytes

Is the count of bytes to allocate from the heap.

ppMemoryRet

Is the memory address where the pointer to the DS space allocated is returned.

HeapAlloc returns status code 4533 (No heap memory available) if the heap does not contain enough contiguous memory available to meet the requirements of the request.

HeapFree. This procedure returns memory to the heap. The procedure call to HeapFree is as follows:

erc = HeapFree(*pMemory*);

where

pMemory

Is the pointer to the block that was previously allocated from the heap.

HeapFree returns status code 4534 ("Invalid heap block") if the offset passed to HeapFree is not the address of a valid heap block.

Logging Messages for Debugging Purposes

To log requests in a trace buffer as they are received and responded to, the common-code module uses the following *Async.lib* procedures:

- LogMsgIn
- LogRespond
- LogRequest

LogMsgIn. This procedure logs the message pointed to by the global variable pRq. LogMsgIn is used by the system service when it receives a message at its exchange.

LogRespond. This procedure logs the response to the request pointed to by the variable *pRq*. LogRespond is used by the system service before it responds to a client request. It also is used by the library procedures SwapContextUser and TerminateContextUser before responding to outstanding client requests.

LogRequest. This procedure logs the request pointed to by *pRq*. LogRequest is called by the BuildAsyncRequest and BuildAsyncRequestDirect procedures before Request and RequestDirect, respectively, are called.

Initializing

Memory initialization is handled by the AllocMemoryInit operation,

AllocMemoryInit. To allocate memory out of the DS space below the last CODE segment, the common-code module uses the procedure AllocMemoryInit. To allocate memory addressable by near pointers AllocMemoryInit obtains the memory from two sources. First, it obtains memory from a memory array in the object module InitAlloc. (For details on InitAlloc, see "Linking Your System Service," later in this section.) Second, when no more space is available, AllocMemoryInit allocates additional memory from the operating system using the ExpandAreaSL operation. (For details on ExpandAreaSL, see the *CTOS Procedural Interface Reference Manual.*)

The procedure call to AllocMemoryInit is as follows:

erc = AllocMemoryInit(*cBytes*, *ppMemoryRet*, *fInit*);

where

cBytes

Is the count of bytes to be allocated.

ppMemoryRet

Is the memory address to which the pointer to the DS space allocated is returned.

fInit

Is TRUE only if the data allocated will be initialized or used before all COED modules have finished executing. (A COED module is an initialization module that is deallocated after use. Setting this flag prevents code yet to be executed from being overwritten. AllocMemoryInit returns status code 26 ("Stray interrupt") if *fInit* is TRUE and the next space to allocate is COED space. To avoid this problem, your program should allocate memory before COED modules have finished executing, and not initialize the memory until after the COED modules have finished executing. In this case, *fInit* can be set FALSE.

All memory is allocated before the call to ConvertToSys. Before the call, however, unused memory is deallocated using the ShrinkAreaSL operation. (For details on ConvertToSys and ShrinkAreaSL, see the *CTOS Procedural Interface Reference Manual.*)

Availability of Asynchronous System Service Files

All the files required for creating an asynchronous system service are contained on the CTOS Development Utilities diskettes, Version 12.0 and later. For a complete list of these files and their contents, see your release documentation.

Linking Your System Service

To Link your program, invoke the **LinkV6** command through the Executive and fill out the command form as shown below:

```
Link V6
 Object modules                  @LinkExample.fls
 Run file                        YourSystemService.run
 [Map file]
 [Publics?]
 [Line numbers?]
 [Stack, Dgroup heap size]
 [Max arrray, data]
 [Min arrray, data]
 [Run file mode]
 [Version]
 [Libraries]                     [Sys]<Sys>Async.lib
 [DS Allocation?]                Yes
 [Symbol file]
 [Copyright notice?]
 [File to append]
 [Linker config file]
```

(For general information on linking programs and details on all the fields in the **LinkV6** command form, see the *Linker/Librarian Manual.*) The field entries for linking your asynchronous system service program are described below.

Object Modules

In the *Object modules* field, type the names of all the modules you are going to link. Because the command line is not long enough to contain the names of all the object modules for this example, the at-file *@LinkExample.fls* is used to contain the module names. (For details on using at-files, see the *CTOS Executive Reference Manual.*) The at-file *LinkExample.fls* contains the following modules:

[Sys]<Sys>Async.lib (InitAlloc LogAsync)
[Sys]<Sys>YourMainModule.obj

Note: *For details on other modules that you may need to link with your program, see your programming language manual.*

InitAlloc and LogAsync

Two of the object modules, InitAlloc and LogAsync, are in the library *Async.lib*. To arrange memory in the proper order, you must list InitAlloc as the first module in the *Object modules* field. For debugging purposes, you can include the module LogAsync. Otherwise, to resolve references, you must enter the name of the dummy module LogAsyncDmy.

Main Program

The third module name shown in the at-file is the name of your main program. This module is produced as a result of compiling your main program source file. Any other modules that form a part of your system service would follow the name of the main program in the at-file.

Run File

In the *Run file* field, type a name for your resulting run file. This run file consists of the module you wrote and all the modules linked with it.

Libraries

In the *[Libraries]* field, type **Async.lib**. *Async.lib* contains all the other modules needed by your system service, including all the asynchronous procedures described in this section.

DS Allocation

In the *DS allocation* field, type **Yes** for DS allocation. As a result, your program code is loaded into memory at a higher address than the program data. This arrangement frees space below your program code for use as a dynamically allocatable area containing data relative to DS. (For additional information on DS allocation, see "Stack Format and Calling Conventions," in *CTOS / Open Programming Practices and Standards*.)

Program Example

The remaining pages in this section show the example system service program.

AsyncService.c

The first module shown is the common-code module *AsyncService.c*. You do not write this code. All you need to do is link it with any system service you write. Comments indicate which portions of the system service you must provide. For example, you will see the following declarations for external functions and variables that you need to provide in your module:

```
/*
   External functions, provided by user.
*/
extern void      InitializeServer (void);
extern ErcType   ServeRequest (void);
extern void      HandleTimer (void);
extern void      InitializeTimer (void);

/*
   External Variables, provided by user.
*/
extern Word  cbHeap;  /* size of service heap */
extern Word  defaultStackSize;  /* size of stack
                                   allocated per context */
extern Word  wOSRel;
extern FlagType  fConvertToSys;
extern FlagType  fRespond;
extern Word  priorityServ;  /* process priority of
                               service */
extern char  rgbPartitionName[];  /* name of partition */
```

Example.c

Example.c is a system service that filters file system requests and writes to a recording file the names and access times of all open files. This module complements the common-code module. If you examine the declarations closely, you will see a variable definition corresponding to each of the externally declared variables in the main module. There is also a function for each external function declaration in the main module.

If your program requires only a subset of the functions declared in the common-code module, you must include a stub for each function you don't need. By doing so, you resolve the reference in the common-code module. For example, the filter program does not require additional initialization beyond that provided by the common-code. To resolve the reference, the filter program includes the stub InitializeServer.

Note: Example.c *does not show how to initialize or handle timers. However, there is provision for calling a HandleTimer procedure in the common-code module. The necessary links for the Timer procedure code are included and identified by comments in* Example.c.

At a minimum, your program requires the function for serving requests. (See the function ServeRequest in Example.c.) In addition to serving system requests for termination, abort, and swapping, your system service serves requests unique to its application. The filter program, for example, intercepts open file requests. The ServeRequest function in this program serves all open file requests as well as all system requests.

To write your main module source code, you can use *Example.c* as a template.

Start.c, Stop.c, and DeInstall.c

As mentioned earlier in this section, *Start.c*, *Stop.c*, and *DeInstall.c* are three other programs that support the filter service. These programs are compiled separately and used by a utility such as the Executive to activate and deactivate the recording process provided by the filter and to carry out deinstallation. *Start.c* and *Stop.c* are used to start and stop the filter's recording function, respectively. *DeInstall.c* is used at deinstallation to vacate and remove the partition containing the filter service. The source to these programs follows *Example.c*.

Example 16-1. AsyncService.c

```
/*
   External Definitions
*/
#define Syslit
#define RqHeaderType
#include "ctostypes.h"

#include <ctype.h>

#define AllocExch
#define ChangePriority
#define Check
#define CheckErc
#define ConvertToSys
#define ErrorExit
#define FatalError
#define GetPartitionHandle
#define GetUserNumber
#define QueryRequestInfo
#define ResetStack
#define Respond
#define ServeRq
#define SetPartitionLock
#define SetPartitionName
#define ShrinkAreaSL
#define Wait
#include <CTOS.Lib.h>

#define FsErc
#define RqErc
#include "Erc.h"

#include <string.h>
```

Example 16-1. AsyncService.c (cont.)

```
/*
   Async.lib Definitions
*/
#define AllocMemoryInit
#define CreateContext
#define HeapInit
#define LogMsgIn
#define LogRespond
#define ResumeContext
#define TerminateAllOtherContexts
#define TerminateContext

#include "async.h"
#include "exdef.h"

/*
   External variables provided by async.lib
*/
extern PointerpToDeallocate;/* set up by InitAlloc */
extern WordcbFree;/* set up by InitAlloc */

/*
   External functions, provided by user.
*/
extern voidInitializeServer (void);
extern ErcTypeServeRequest (void);
extern voidHandleTimer (void);
extern void InitializeTimer (void);

/*
   External Variables, provided by user.
*/
extern WordcbHeap;/* size of service heap */
extern WorddefaultStackSize;/* size of stack allocated per
context */
extern WordwOSRel;
extern FlagTypefConvertToSys;
extern FlagTypefRespond;
extern WordpriorityServ;/* process priority of service */
extern charrgbPartitionName[];/* name of partition */
```

Example 16-1. AsyncService.c (cont.)

```
/*
   PUBLIC Variables required by the AsyncService procedures:
*/
/* request codes served by this service */
Word rgwRqs[10] = {
   rcAbortExample,
   rcChangeUserNumExample,
   rcDeinstallExample,
   rcOpenFile,
   rcOpenFileLL,
   rcStartRecord,
   rcStopRecord,
   rcSwapExample,
   rcTerminateExample,
   rcReOpenFile};
Word nRqCodes = 10;/* number of request codes served */
Word rgwOldExch[10];/* temp. storage of old exch values */

ErcType ercAsync;
/*
   The erc from BuildAsyncRequest or BuildAsyncRequestDirect
   is returned here.
*/

ExchType exchServ;
/*
   This is the exchange where the service waits in
   its WaitLoop.  This variable is stored in the exchResp
   field of the Request block that is built by
   BuildAsyncRequest or BuildAsyncRequestDirect.
*/

RqHeaderType *pRq;
/*
   pRq points to the current client request. It is set by
   the service in its WaitLoop by the call to Wait:

      erc = Wait(exchServ, &pRq);
```

Example 16-1. AsyncService.c (cont.)

```
   It is saved in the current Context Control Block by
   BuildAsyncRequest.  It is restored by ResumeContext.
*/

PointerpZero = 0;
WordsaveSpBp[2] = {0, 0};/* storage locations for sp & bp */
Word*pSave;
Pointer*pbHeap;/* pointer to allocated heap storage */

void FatalServerError (ErcType erc)
{
   FatalError (erc) /* Kill the system service */;
}
ErcType DeinstallServer (void)
{
   /*
      This function is invoked as a result of deinstall
      request.
   */
   typedef struct {
      RqHeaderType rqhdr;
      Word *pPhRet;
   } DeinstallType;
   ErcType erc;
   Word    i;
   DeinstallType*prx;
   RqHeaderType*pRqBlk;

   prx = (DeinstallType*) pRq;
   /*
      Unserve all requests by serving them to the exchange
      previously served.
   */
   for (i = 0; i < nRqCodes; i++)
   {
      erc = ServeRq (rgwRqs[i], rgwOldExch[i]);
      if (erc != ercOK)
         FatalServerError (erc);
   }
```

Example 16-1. AsyncService.c (cont.)

```
   /*
      Terminate all contexts except for the current context.
   */
   erc = TerminateAllOtherContexts (ercOK);
   if (erc != ercOK)
      FatalServerError (erc);

   /*
      Reject all further incoming requests to service's
      exchange.
   */
   while (erc == ercOK)
   {
      erc = Check (exchServ, &pRqBlk);
      if (erc == ercOK)
      {
         pRqBlk->ercRet = ercServiceNotAvail;
         LogRespond (pRqBlk);
         erc = Respond (pRqBlk);
         if (erc != ercOK)
            FatalServerError(erc);
      }
   }

   /*
      Return partition handle to calling program so that the
      calling program can vacate the partition and remove
      it.
   */
   ercAsync = SetPartitionLock (FALSE);
   return GetPartitionHandle (&rgbPartitionName,
                  strlen(rgbPartitionName)prx->pPhRet);
}
```

Example 16-1. AsyncService.c (cont.)

```
/*
   HandleRequest: The request pointed to by pRq is a new
   Request or a TRB.  Given a request code, call the routine
   which processes the request.  NOTE that this procedure is
   NOT REENTRANT.  After the call to CreateContext, its
   stack pointer has changed, so it cannot access any local
   variables on the stack.
*/
void HandleRequest (void)
{
   if (pRq->rqCode == 0)
   {  /*
         rqCode = 0 indicates a timer request block
      */
      HandleTimer ();
      return;
   }

   ercAsync = CreateContext (defaultStackSize,
                             pRq->userNum);
   if (ercAsync == ercOK)
   {  /*
         Process request in user-supplied function
      */
      pRq->ercRet = ServeRequest ();
   }
   else
   {  /*
         Context could not be started because no heap space
      */;
      pRq->ercRet = ercAsync;
   }

   if (fRespond) {
      LogRespond (pRq);
      ercAsync = Respond (pRq);
   }
   fRespond = TRUE;
   ercAsync = TerminateContext ();
```

Example 16-1. AsyncService.c (cont.)

```
   /*
      Only returns if error -- normally calls WaitLoop
   */
   FatalServerError (ercAsync);
}
void Initialize(void)
{
   /*
      This function is called to initialize the service.
   */
   Word i;
   Word userNumServ;

   pSave = &saveSpBp[0]; /* establish pointer to sp/bp save
                                        area */

   /*
      Allocate exchange for service.
   */
   CheckErc (AllocExch (&exchServ));

   /*
      Set process priority of service.
   */
   CheckErc (ChangePriority (priorityServ));

   /*
      Test Requests;
   */
   if (wOSRel < 10)
      nRqCodes--;

   for (i = 0; i < nRqCodes; i++)
      /*
         Test each request enumerated in rgwRqs[] to verify
         that it also exists in the Request.sys file read at
         boot time.
      */
      CheckErc (QueryRequestInfo (rgwRqs[i], &rgwOldExch[i],
                                        2));
```

Example 16-1. AsyncService.c (cont.)

```
/*
   Allocate memory from DS space to be used for the heap.
*/
CheckErc (AllocMemoryInit (cbHeap, &pbHeap, FALSE));
CheckErc (HeapInit (cbHeap, pbHeap));
/*
   Allocate any other memory required for tables, etc
   from DS space.
*/
if (cbFree != 0)
   /*
      Free leftover memory used by the initialization
      code.
   */
   CheckErc (ShrinkAreaSL (pToDeallocate, cbFree));

/*
   Service specific initialization.
*/
InitializeServer ();

/*
   Initialize Timer Request Block as necessary.
*/
InitializeTimer ();
if (fConvertToSys)
   CheckErc (ConvertToSys ());

/*
   Serve Requests
*/
for (i = 0; i < nRqCodes; i++)
{
   /*
      Serve each request code enumerated for this service.
   */
   CheckErc (ServeRq (rgwRqs[i], exchServ));
}
```

Example 16-1. AsyncService.c (cont.)

```
   /*
      Call ErrorExit (ercOK) to 'become' part of OS.
   */
   ErrorExit (ercOK);

   ercAsync = GetUserNumber (&userNumServ);
   /*
      SetPartitionName must come after ConvertToSys.
   */
   ercAsync = SetPartitionName ((userNumServ & 0xFF),
              &rgbPartitionName, strlen (rgbPartitionName));
}
/*
   WaitLoop: Main process loop.
   It waits for a request or a response to come in.
*/
void WaitLoop (void)
{
   ErcTypeerc;

   erc = ResetStack (offsetof (pSave));
   FOREVER
   {
      erc = Wait (exchServ, &pRq);
      LogMsgIn ();
      if (erc == ercOK)
      {
         /*
            If the response exchange in the request block
            equals this service's service exchange, then
            the request was sent by this service.  If the
            request code is 0, then the request is a timer
            request block.
         */
```

Example 16-1. AsyncService.c (cont.)

```
        if ((pRq->exchResp==exchServ) && (pRq->rqCode != 0))
           /*
              The request in rq is a response to an
              asynchronous request that we created
              and sent out. Handle the response.
           */
           erc = ResumeContext ();
           /*
              Only returns if error was detected.
           */
        else
           /*
              We have a new Request or a Timer
              Request Block.
           */
           HandleRequest ();
     }
     if (erc != ercOK)
        FatalServerError (erc);
  }
}

void AsyncServer (void)
{
  Initialize ();
  WaitLoop ();
}
```

Example 16-2. Example.c

```
/*
   External definitions.
*/
#define RqHeaderType
#define Syslit
#define sdType
#define Sublit
#define TRBType
#include "ctostypes.h"

#include <ctype.h>

#define ChangeFileLength
#define CheckErc
#define CloseFile
#define CreateFile
#define CurrentOsVersion
#define ForwardRequest
#define GetDateTime
#define GetUserNumber
#define NlsStdFormatDateTime
#define OpenFile
#define QueryRequestInfo
#define RgParam
#define SetFileStatus
#include <CTOS.Lib.h>

#define FsErc
#define RqErc
#include "Erc.h"

#define BuildAsyncRequest
#define BuildAsyncRequestDirect
#define CheckContextStack
#define CreateContext
#define DebugTrap
```

Example 16-2. Example.c (cont.)

```
#define HeapAlloc
#define HeapFree
#define SwapContextUser
#define TerminateContext
#define TerminateContextUser
#include "Async.h"

#include <string.h>
#include <stdio.h>
#include "ExDef.h"

#define All
#include "ExRqBlk.h"

/*
   External functions provided by user.
*/
extern void AsyncServer (void);
extern ErcType DeinstallServer (void);
extern void FatalServerError (ErcType erc);

/*
   Constants
*/
#define NDATESIZE19
#define LF    0x0A
#define rcChangeFileLength13
#define rcGetFileStatus8
#define rcWrite36

/*
   External variables.
*/
extern RqHeaderType*pRq;
extern ErcTypeercAsync;
extern PointerpZero;
```

Example 16-2. Example.c (cont.)

```
/*
     Public variables.
*/
char        rgbPartitionName[] = "Example";
char        rgbPad[] = "          \n";
char        rgbMissedFiles[] = "Could not open recording
                          file, records discarded.\n";
Byte        bReadWritePro = 15;/* Protection = R/W without
                                  password */
FlagType    fConvertToSys = TRUE;/* FALSE for debugging */
FlagType    fDebug = FALSE;
FlagType    fRespond = TRUE;
Word        wOSRel, wOSRev;
TRBType     TimeRq;          /* Timer Request Block */

/*
     The following variables must be set before main ()
     calls AsyncServer ().  If any of these values need to
     be computed at runtime, do it in
     InitializeBeforeAsync ().
*/
Word        cbHeap = 6000;         /* Total size of heap */
Word        defaultStackSize = 1200;/* Stack space for each
                                       context */
Word        priorityServ = 0x20;/* Process priority of
                                   service */

/*
     Module variables.
*/
char        rgbRecordFile[255];
Word        cbRecordFile;
char        zDefaultRecordFile[]="[sys]<sys>OpenFile.log\0";
Pointer     pbBuffer[2];     /* 2 buffer pointers for double
                                buffering */
char        Buffer1[1024], Buffer2[1024];
                          /* buffers for I/O */
Word        iActive = 0;          /* index of active buffer */
Word        iBuf = 0;        /* buffer data pointer in active
                                buffer */
```

Example 16-2. Example.c (cont.)

```
char        *pWork;                 /* pointer to data in active
                                       buffer */
FlagTypefRecording = TRUE;          /* recording boolean */
ExchTypeexchOpenFile, exchOpenFileLL, exchReOpenFile;
ErcType AddToBuffer (sdType *pSd)
{
     /*
          This function is invoked for each open file
          request seen by this service.  The file name and
          the date/time it was referenced is placed in the
          current buffer.  If there is insufficient space
          in the buffer, DumpBuffer() is invoked to write
          the buffer to the recording file.
     */
     ErcTypeerc;
     Word  len;
     DWord dateTime;
     externErcType DumpBuffer (void);

     if (fRecording)
     {
          /*
               If not enough space is left in the buffer
               for this entry, dump the buffer to the file.
          */
          if (iBuf+pSd->cb+NDATESIZE+2 > sizeof (Buffer1))
          {
               erc = DumpBuffer ();
               if (erc != ercOK)
               {
                    /*
                         Could not open file, try to
                         record event.
                    */
                    strcpy(pWork, (char*)&rgbMissedFiles);
                    pWork += strlen (rgbMissedFiles);
                    iBuf += strlen (rgbMissedFiles);
               }
          }
```

Example 16-2. Example.c (cont.)

```
        /*
             Get date and time of reference
        */
        erc = GetDateTime (&dateTime);
        if (erc != ercOK)
             return (erc);

        /*
             Expand it to human-readable form
        */
        erc = NlsStdFormatDateTime (NULL, 16, dateTime,
                  pWork, sizeof (Buffer1) - iBuf, &len);
        if (erc != ercOK)
             return (erc);

        /*
             Put date stamp and filename in buffer
        */
        pWork += len;
        iBuf += len;
        *pWork++ = ' ';
        iBuf += 1;
        strncpy (pWork, pSd->pb, pSd->cb);
        pWork += pSd->cb;
        iBuf += pSd->cb;
        *pWork++ = LF;          /* add line feed to
                                   filename/timestamp */
        iBuf += 1;
    }

    if (CheckContextStack () != ercOK)
    {
        /*
             defaultStackSize is too small!!
        */
        DebugTrap ();
    }
}
```

Example 16-2. Example.c (cont.)

```
ErcType DumpBuffer (void)
{
      /*
            This function is invoked whenever
            (1) a buffer fills,
            (2) recording is turned off,
            (3) the service is deinstalled.
      */
      Word len;
      FhTypefh;
      ErcTypeerc;
      DWordlfa;
      Pointerp;

      /*
            Space fill between last data and end of buffer
      */
      while (iBuf < sizeof (Buffer1))
      {
            len = sizeof (rgbPad);
            if (len > (sizeof (Buffer1) - iBuf))
                  len = sizeof (Buffer1) - iBuf;
            strncpy (pWork, rgbPad, len);
            pWork += len;
            iBuf += len;
      }
      --pWork;
      *pWork++ = LF;

      /*
            Assign a new buffer for output.
      */
      p = pbBuffer[iActive];
      iActive = (iActive + 1) & 1;
      pWork = pbBuffer[iActive];
      iBuf = 0;
```

Example 16-2. Example.c (cont.)

```
/*
     Open the file in mode modify
*/
erc = BuildAsyncRequestDirect (&fh, &rgbRecordFile,
                cbRecordFile, (DWord)0, 0, modeModify,
                rcOpenFileLL, exchOpenFileLL);
if (erc == ercOK)
{
     /*
          Change the file length
     */
     erc = BuildAsyncRequest (fh, 0, &lfa, 4,
                          rcGetFileStatus);
     if (erc != ercOK)
          return (erc);
     erc = BuildAsyncRequest (fh, lfa + sizeof
                (Buffer1), rcChangeFileLength);
     if (erc != ercOK)
          return (erc);

     /*
          Write the data
     */
     erc = BuildAsyncRequest (fh, p, sizeof (Buffer1),
                          lfa, &len, rcWrite);
     if (erc != ercOK)
          return (erc);

     /*
          Close the file
     */
     erc = BuildAsyncRequest (fh, rcCloseFile);
}
```

Example 16-2. Example.c (cont.)

```
    if (CheckContextStack () != ercOK)
    {
        /*
            defaultStackSize is too small!!
        */
        DebugTrap ();
    }

    return (erc);
}
/*
    HandleTimer: The request in pRq is a Timer Request
    Block.  NOTE that this procedure is NOT REENTRANT.
    After the call to CreateContext, its stack pointer has
    changed, so it cannot access any local variables on
    the stack.
*/
void HandleTimer (void)
{
    /*
        The timer is not used in this example.  This
        function is provided only to resolve the external
        reference made in the Asynchronous Service code.
    */
    ercAsync = CreateContext (defaultStackSize, 0);
    if (ercAsync != ercOK)
    {   /*
            Context could not be started because no heap
            space
        */
        TimeRq.cEvents = 0;/* Re-enable timer */
    }
    else
    {   /*
            Context started OK.
            Insert timer processing code here.
        */
```

Example 16-2. Example.c (cont.)

```
            ercAsync = TerminateContext ();
            /*
                  Only returns if error - normally calls
                  WaitLoop
            */
            FatalServerError (ercAsync);
      }
}

/*
      Early service initialization, service heap not yet
      initialized.
*/
void InitializeBeforeAsync (void)
{
      ErcTypeerc;
      FhTypefh;
      sdTypesd;

      /*
            Process '[Recording file]' parameter
      */
      strcpy (rgbRecordFile, zDefaultRecordFile);
      cbRecordFile = strlen (zDefaultRecordFile);
      erc = RgParam (1, 0, &sd) /* log file name */;
      if (erc == ercOK)
      {
            strncpy (&rgbRecordFile[0], sd.pb, sd.cb);
            cbRecordFile = sd.cb;
            rgbRecordFile[sd.cb] = 0;
      }

      erc = CreateFile (&rgbRecordFile, cbRecordFile, pZero,
                  0, 1024);
      CheckErc (OpenFile (&fh, &rgbRecordFile, cbRecordFile,
                  pZero, 0, modeModify));
      CheckErc (ChangeFileLength (fh, 0));
      CheckErc (SetFileStatus (fh, 2, &bReadWritePro, 1));
      CheckErc (CloseFile (fh));
```

Example 16-2. Example.c (cont.)

```
    /*
         Establish array of buffer pointers for double
         buffering.
    */
    pbBuffer[0] = &Buffer1[0];
    pbBuffer[1] = &Buffer2[0];
    pWork = pbBuffer[iActive = 0];

    /*
         Get OS Version; ReOpenFile request does not exist
         on real mode Workstation OS's.
    */
    CheckErc (CurrentOsVersion (&wOSRel, &wOSRev));

    /*
         Save exchange values for forwarding;
    */
    CheckErc (QueryRequestInfo (rcOpenFile, &exchOpenFile,
                                2));
    CheckErc (QueryRequestInfo (rcOpenFileLL,
                                &exchOpenFileLL, 2));
    if (wOSRel >= 10)
         CheckErc (QueryRequestInfo (rcReOpenFile,
                                &exchReOpenFile, 2));

    /*
         Output sign-on message
    */
    printf ("\nFile Access Recorder installed.\n");
}
/*
    Service-specific initialization, called before
    ConvertToSys.  The Service Heap is initialized.
*/
```

Example 16-2. Example.c (cont.)

```
void InitializeServer (void)
{
     /*
          No additional initialization is required by this
          example.
     */
}

/*
     Timer initialization, called before ConvertToSys.
     The Service Heap is initialized.
*/
void InitializeTimer (void)
{
     /*
          Timer unused in this example.  This function is
          provided to satisfy external reference made by
          the Asynchronous Service code.
     */
}
/*
     Service-specific request processing procedure.
*/
ErcType ServeRequest (void)
{
     ErcTypeerc,
               ercDiscard;
     sdTypesd;

     switch (pRq->rqCode)
     {
          case rcAbortExample:
          {
               AbortExampleType *prx;

               prx = (AbortExampleType *) pRq;
               erc = TerminateContextUser
                    (prx->rqhdr.userNum, ercOK);
               break;
          }
```

Example 16-2. Example.c (cont.)

```
case rcChangeUserNumExample:
{
     ChangeUserNumExampleType *prx;

     prx = (ChangeUserNumExampleType *) pRq;
     break;
}

case rcDeinstallExample:
{
     DeinstallExampleType *prx;

     prx = (DeinstallExampleType *) pRq;
     erc = DumpBuffer ();
     fRecording = FALSE;
     DeinstallServer ();
     break;
}

case rcOpenFile:
{
     OpenFileType *prx;

     prx = (OpenFileType *) pRq;
     sd.cb = prx->cbFileSpec;
     erc = HeapAlloc (sd.cb, &sd.pb);
     if (erc == ercOK)
          strncpy (sd.pb, prx->pbFileSpec,
                          sd.cb);

     ercDiscard = ForwardRequest (exchOpenFile,
                          pRq);

     if (erc == ercOK)
     {
          ercDiscard = AddToBuffer (&sd);
          ercDiscard = HeapFree (sd.pb);
     }
     fRespond = FALSE;
     break;
}
```

Example 16-2. Example.c (cont.)

```
case rcOpenFileLL:
{
      OpenFileLLType *prx;

      prx = (OpenFileLLType *) pRq;
      sd.cb = prx->cbFileSpec;
      erc = HeapAlloc (sd.cb, &sd.pb);
      if (erc == ercOK)
            strncpy(sd.pb, prx->pbFileSpec,
                        sd.cb);

      ercDiscard = ForwardRequest (exchOpenFileLL,
                              pRq);

      if (erc == ercOK)
      {
            ercDiscard = AddToBuffer (&sd);
            ercDiscard = HeapFree (sd.pb);
      }
      fRespond = FALSE;
      break;
}

case rcReOpenFile:
{
      ReOpenFileType *prx;

      prx = (ReOpenFileType *) pRq;
      sd.cb = prx->cbFileSpec;
      erc = HeapAlloc (sd.cb, &sd.pb);
      if (erc == ercOK)
            strncpy (sd.pb, prx->pbFileSpec,
                        sd.cb);

      ercDiscard = ForwardRequest (exchReOpenFile,
                        pRq);
```

Example 16-2. Example.c (cont.)

```
        if (erc == ercOK)
        {
                ercDiscard = AddToBuffer (&sd);
                ercDiscard = HeapFree (sd.pb);
        }
        fRespond = FALSE;
        break;
}

case rcStartRecord:
{
        StartRecordType *prx;

        prx = (StartRecordType *) pRq;
        fRecording = TRUE;
        break;
}

case rcStopRecord:
{
        StopRecordType *prx;

        prx = (StopRecordType *) pRq;
        erc = DumpBuffer ();
        fRecording = FALSE;
        break;
}

case rcSwapExample:
{
        SwapExampleType *prx;

        prx = (SwapExampleType *) pRq;
        erc = SwapContextUser (prx->rqhdr.userNum);
        break;
}
```

Example 16-2. Example.c (cont.)

```
            case rcTerminateExample:
            {
                  TerminateExampleType *prx;

                  prx = (TerminateExampleType *) pRq;
                  erc = TerminateContextUser
                        (prx->rqhdr.userNum, ercOK);
                  break;
            }

            DEFAULT:
            {
                  /*
                        unrecognized request code
                  */
                  return (ercNoSuchRc);
            }
      }
      return (ercOK);
}
/*
      Main program.
*/
void main (void)
{
      InitializeBeforeAsync ();
      AsyncServer ();
}
```

Example 16-3. Start.c

```
#define Syslit
#include "ctostypes.h"

#include "ctype.h"

#define CheckErc
#include "CTOS.Lib.h"


#define StartRecord
#include "ExFunc.h"

/*
     Program used to enable file open recording of File
     Access Recorder service.
*/
void main (void)
{

     CheckErc (StartRecord ());
}
```

Example 16-4. Stop.c

```
#define Syslit
#include "ctostypes.h"

#include "ctype.h"

#define CheckErc
#include "CTOS.Lib.h"


#define StopRecord
#include "ExFunc.h"
/*
     Program used to disable file open recording of File
     Access Recorder service.
*/
void main (void)
{
     CheckErc (StopRecord ());
}
```

Example 16-5. Deinstall.c

```
#define Syslit
#include "ctostypes.h"

#include <ctype.h>

#define CheckErc
#define RemovePartition
#define VacatePartition
#include <CTOS.Lib.h>


#define DeinstallExample
#include "ExFunc.h"

#include <stdio.h>

/*
     Program used to remove File Access Recorder service
     from memory.
*/
void main (void)
{
     Word PartitionHandle;

     CheckErc (DeinstallExample (&PartitionHandle));
     CheckErc (VacatePartition (PartitionHandle));
     CheckErc (RemovePartition (PartitionHandle));
     printf ("\nFile Access Recorder removed.\n");
}
```

Section 17
CD-ROM Service

Overview

The CD-ROM Service provides access to CD-ROM (compact disc read-only memory), a read-only media used for storing documents, databases, audio, or combinations of data and audio elements. The data portion of the CD-ROM Service supports both the ISO-9660 and High Sierra standards. The CD-ROM Service can be installed on a workstation with at least one CD-ROM disc drive. Within a cluster, the workstation can be either a server or a client workstation. If installed on the server, client workstations of the server can also use the CD-ROM Service.

To use the operations described in this section, first use the **Install CDROM Service** command to install the CD-ROM Service on your workstation, as described in the *CTOS Executive Reference Manual* and the *CTOS System Administration Guide*. This command installs the system service and also sets the maximum number of CD-ROM files that can be open by all users at one time, as well as the maximum number of users that can access all CD-ROM drives connected to the workstation that is running the CD-ROM Service.

Requirements

The CD-ROM Service operates on any workstation that uses one of the following versions of the operating system (or a subsequent version):

- CTOS III 1.0
- CTOS/VM
- CTOS II 3.3
- BTOS II

It requires at least one model CD-001/2 or B25-CDC/X CD-ROM module or compatible SCSI CD-ROM disc drive.

Functional Groups of Operations

The following sections offer a brief description of the CD-ROM Service operations. See the *CTOS Procedural Interface Reference Manual* for complete descriptions of these operations.

Volume Information

CdDirectoryList

Returns a list of all directory paths for a CD-ROM disc. (See the section on "File Structure: Hierarchical vs. Flat," later in this section.)

CdGetDirEntry

Returns directory record information, in either ISO or High Sierra format, for a specified path.

CdGetVolumeInfo

Returns specific information for a particular volume (the primary volume descriptor, copyright file name, abstract file name, and bibliographic file name).

CdSearchFirst

For a given file specification (possibly with wild cards), returns the directory record and path information for the first matching file on the disc.

CdSearchNext

If wild cards were used in a previous CdSearchFirst operation, finds additional files that match the wild-card specification.

CdSearchClose

Closes an open CD-ROM search session.

CdVerifyPath

Verifies the existence of the specified path.

CdVersionRequest

Returns information concerning the software version level, hardware environment, number of CD-ROM drives accessible by the service, and the device name assigned during installation. The default is *[CDROM]*.

Status

CdControl

Controls disc eject and door locking mechanisms. In addition, returns device status, location of head, sector size, and volume size.

Read File

CdOpen

Opens an existing CD-ROM file or device and returns a file or device handle.

CdRead

Transfers a specified number of bytes from CD-ROM to memory.

CdClose

Closes an open CD-ROM file or device.

Audio

CdAudioCtl

Controls the audio playback channels in the CD-ROM disc drive, including selection of output channels and volume setting and control of audio playback, pause, and stop. Also provides information on the disc as a whole, on particular tracks, and on the current Q-channel address.

Miscellaneous (Rarely Used)

CdAbsoluteRead

Performs a physical read of a specified number of blocks from CD-ROM to memory. This read is performed without regard to the specific file structure used by the disc creator.

CdServiceControl

Deinstalls the CD-ROM Service.

Standard File Formats

Data can be stored on a CD-ROM disc in one of two standard file formats: ISO-9660 or High Sierra. Your first step in programming for the CD-ROM is to find out which format the target disc uses. This section includes a code excerpt showing this process, as well as descriptions of the formats for both ISO and High Sierra directory records and primary volume descriptors.

Operations that require knowledge of disc format are

CdGetDirEntry
CdGetVolumeInfo

Determining the File Format Using CdGetVolumeInfo

Use the CdGetVolumeInfo request, function 4, to read the primary volume descriptor for the disc. The complete primary volume descriptor is 2048 bytes. However, the information you need to determine the file format is found in the first 14 bytes of this structure. In the High Sierra primary volume descriptor, the first 14 bytes are as follows:

Bytes	Field	Content
1-8	Volume Descriptor LBN	numeric value
9	Volume Descriptor Type	numeric value
10-14	Volume Structure Standard Identifier	CDROM

In the ISO primary volume descriptor, the first 6 bytes contain the relevant information:

Bytes	Field	Content
1	Volume Descriptor Type	numeric value
2-6	Standard Identifier	CD001

The following code shows the use of a CdGetVolumeInfo request to determine the format of the target CD-ROM disc. To conserve memory, specify 14 bytes instead of 2048 bytes in the CdGetVolumeInfo request if you are concerned only with the standard identifier field.

```
unsigned long int     CDh;
unsigned char         *CdBuff;
char                  rgbIsoIdentifier[] = "CD001";
char                  rgbHsgIdentifier[] = "CDROM";

CheckErc(CdOpen(0,From->pb, From->cb, NULL, 0, &CDh));
CheckErc(CdGetVolumeInfo(CDh, CdBuff, 2048, "&"wDataRet,
   4));
i = ULCmpB(CdBuff+1, &rgbIsoIdentifier, 5);
if (i != 0xFFFF) {
   i = ULCmpB(CdBuff+9, &rgbHsgIdentifier, 5);
   if (i !=- 0xFFFF) {
      free(CdBuff);
      exit(0);
   }
   else
      fFormatType = HSG_DISC;
}
else
   fFormatType = ISO_DISC;
```

Other Uses of CdGetVolumeInfo

After you have determined the disc's format, you can use the other functions of the CdGetVolumeInfo request as described below:

Function Number	Information Retrieved
1	Copyright file name ISO copyright file name is 37 bytes High Sierra copyright file name is 32 bytes
2	Abstract file name (requires 38 bytes) ISO abstract filename is 37 bytes High Sierra abstract file name is 32 bytes
3	Bibliographic file name (requires 38 bytes). ISO bibliographic file name is 37 bytes. High Sierra bibliographic file name is 32 bytes

File Structure: Hierarchical vs. Flat

The CD-ROM Service recognizes two types of file specifications: the CTOS file specification and the hierarchical file specification. The CTOS file specification is used for "flat" file structures (described below). The backslash (\) file specification is used for hierarchical file structures and can also be used for flat file structures.

CTOS File Specification (for Flat File Structures)

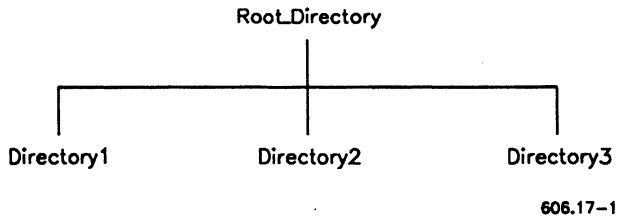
A CTOS file specification has the following form:

{node} [device_name] <directory_name> file_name

The node name is optional and if omitted the target is assumed to be local.

In the directory tree shown in Figure 17-1, a CTOS file specification could be used to access files in *Directory1*, *Directory2*, and *Directory3*. Because this directory tree is only one level deep, it can be described as a "flat" file structure.

Figure 17-1. Flat File Structure



For example:

[CDROM0] <Directory2> MyFile

A CTOS file specification could not be used to access the file contained in the root directory in this example.

Backslash File Specification (for Hierarchical File Structures)

A backslash file specification could also be used to access any of the files contained in the directory tree shown in Figure 17-1, including the files in the root directory. For example:

[CDROM0]\Directory2\MyFile

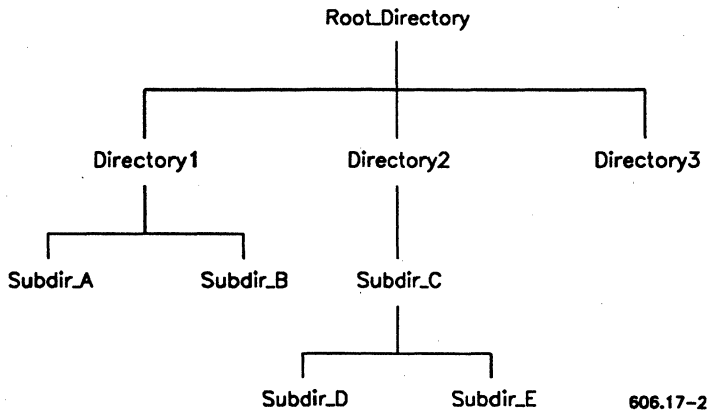
Equivalent to the CTOS specification
[Root_Directory]<Directory2>MyFile

[CDROM0]\YourFile

To access a file in the root directory

In addition, a backslash file specification can be used to access files in any of the directories or subdirectories shown in Figure 17-2.

Figure 17-2. Hierarchical File Structure



For example:

[CDROM0]\Directory2\Subdir_C\Subdir_E\AprilAccounts.txt

Obtaining the Directory List

Use the CdDirectoryList operation to obtain the directory list for a particular CD-ROM disc. The syntax for CdDirectoryList is

CdDirectoryList(pbDevSpec, CbDevSpec, pbBuff, CBBUFF, PSDATARET): *ercType*

where

pbDevSpec
cbDevSpec

Describe a character string of the form *{node} [devname].*

pbBuff
cbBuff

Describe the buffer to which the directory list is returned.

psDataRet

Is the memory address where the count of bytes in the directory list is returned.

The directory list entries (returned to *pbBuff/cbBuff*) are character strings in which the first byte is the size of the string (sbType). The directories in a hierarchical file structure are separated by the backslash character (5C). For example, the directory list buffer for the example directory tree shown in Figure 17-2 is shown below.

Offset (HEX)	Directory List Buffer (HEX)
00	0a 44 49 52 45 43 54 4F 52 59 31 13 44 49 52 45
10	43 54 4F 52 59 31 5C 53 55 42 44 49 52 5F 41 13
20	44 49 52 45 43 54 4F 52 59 31 5C 53 55 42 44 49
30	52 5F 42 0a 44 49 52 45 43 54 4F 52 59 32 13 44
40	49 52 45 43 54 4F 52 59 32 5C 53 55 42 44 49 52
50	5F 43 1C 44 49 52 45 43 54 4F 52 59 32 5C 53 55
60	42 44 49 52 5F 43 5C 53 55 42 44 49 52 5F 44 1C
70	44 49 52 45 43 54 4F 52 59 32 5C 53 55 42 44 49
80	52 5F 43 5C 53 55 42 44 49 52 5F 45 0a 44 49 52
90	45 43 54 4F 52 59 33

Example: Using CdDirectoryList

The following example illustrates the use of the CdDirectoryList request. The four examples included in this section can be found on the floppy disk supplied with this manual. The source code, along with all other necessary executable files, are provided in archive format. To extract the example files from the archive, invoke the **Restore** command and fill out the form as follows:

```
Restore
 [Archive file]                    examples
 [File list from (<*>*)]           <CdTest>*
 [File list to (<*>*)]             [YOUR VOLUME]
                                   <YOUR DIRECTORY>*
 [Overwrite ok?]                   yes
 [Confirm each?]
 [Sequence number]
 [Merge with existing file?]
 [List files only?]
 [Print file]
 [Suppress confirmation?]
```

To build the example programs, first compile the C source files. All of the examples have been designed to link in the same manner. To link Example 1, use the Link V6 command as follows:

```
Link V6
 Object modules                    example1.obj utl.obj
                                   start.obj
 Run file                          example1.run
 [List file]
 [Publics?]
 [Line numbers?]
 [Stack size]
 [Max memory array size]
 [Min memory array size]
 [System build?]                   protected
 [Version]                         Your_Version_Here
 [Libraries]                       [Sys]<Sys>Ctos.lib
                                   [Sys]<Sys>CtosToolKi
 t.lib
 [DS allocation?]
 [Symbol file]
 [Copyright notice?]
 [File to append]
 [Debug?]
```

To link the other examples, replace all references to example1 with example*n*, where *n* is the example number.

Use the Run command to invoke the program. For example:

```
Run
 Run file          example1.run
 [Case]
 [Command]
 [Parameter 1]
 [Parameter 2]
```

Examples 1, 2, and 4 require only the appropriate run file name. Example 3 requires additional parameters, as described later in this section.

Example 17-1. CdDirectoryList Example

```
/* The following code illustrates the use of the
 * CdDirectoryList Request.  This is example1.
 */

#define offsetof(X)              (((Word *)&(X))[0])
typedef unsigned short int       USI;          /* word  */
typedef unsigned char            UC;           /* byte  */
typedef unsigned long int        ULI;          /* dword */
typedef void _far *              Pointer;

typedef struct {
   UC   cb;
   char rg[255];
}  sbType;

char  rgbDevName[]     = "[CDROM0]";
char  rgbDirList[]     = "DirectoryList";
char  rgbFiveSpaces[] = "     ";

/* Obtains the DirectoryList for the installed CD-ROM
 * disc.
 */
```

Example 17-1. CdDirectoryList Example (cont.)

```
USI GetDirList(void)
{
USI       wDataRet;
sbType    *DirBuff;
USI       i;
USI       erc;

   erc = AllocMemorySL(4096, &DirBuff);
   if (erc != ercOK)
      return erc;

   NewLine();
   OutputToVid0(&rgbDirectoryList,
      sizeof(rgbDirectoryList)-1);

   NewLine();
   erc = CdDirectoryList(&rgbDevName sizeof(rgbDevName)-
            1, DirBuff, 2048, &wDataRet);
   if (erc != ercOK)
      return erc;

   /* Display all directory paths */
   i = 0;
   while (i < wDataRet)  {
    OutputToVid0(&rgbFiveSpaces, sizeof(rgbFiveSpaces)- 1);
    OutputToVid0(&DirBuff->rg, DirBuff->cb);
    NewLine();
    i = i + DirBuff->cb + 1;
    offsetof(DirBuff) = offsetof(DirBuff) + DirBuff->cb + 1;
   }
   return erc;
}
```

Searching for Files

The following example shows the use of the CD-ROM search operations (CdSearchFirst, CdSearchNext, and CdSearchClose) as well as the CdVersion operation.

CdSearchFirst finds the first file on the CD-ROM disc that matches the file specification and returns the directory record and path information for that file. CdSearchFirst performs an implicit CdOpen and returns the search file handle and full file specification for the first matching file. (Although this example does not use it, the directory record for the matching file is also returned.) The file specification can be in either standard CTOS format or backslash format, as described earlier in this section, and it can include wild-card characters.

If the CdSearchFirst operation uses wild-card characters, additional files that match the specification can be found by calling CdSearchNext until status code 15715 (No such CD-ROM file) is returned. CdSearchNext returns directory information and path information for each matching file.

The CdSearchClose operation performs an implicit CdClose, freeing the search file handle and search buffers and closing the CD-ROM disc.

The following example uses CdVersion to display the device names with which the CD-ROM Service was installed, as well as the revision level of the installed CD-ROM Service and the hardware environment. (Note that although the CdVersion operation requires a device specification, this parameter is merely a placeholder to be sure the request is routed correctly and is not verified by the operating system. The actual device names are returned by CdVersion. In this example, the device names are updated if nondefault names were supplied.)

Example: File Search

Example 17-2. CD Search Example

```
/* The following code illustrates the use of the
 * CdSearchFirst, CdSearchNext, CdSearchClose, and
 * CdVersion Requests.  This is example2*/

#define offsetof(X)              (((Word *)&(X))[0])

typedef unsigned short int       USI;
typedef unsigned char            UC;
typedef unsigned long int        ULI;
typedef void _far *              Pointer;

typedef struct {
   UC   cb;
   char rg[255];
}  sbType;

void movb (void *src, void *dst, size_t cb);
void setb (char ch, void *dst, size_t cb);

char    rgbDevName[]     = "[CDROM0]";
UC      cbDevName        = 8;
char    rgbSearchSpec[]  = "<*>*";
char    rgbSearchClose[] = "CdSearchClose
        (Request 81BEh)";
char    rgbSearchFirst[] = "CdSearchFirst
        (Request 81BFh)";
char    rgbSearchNext[]  = "CdSearchNext
        (Request 81C0h)";
char    rgbSearchFor[]   = "     Searching for ";
char    rgbFiveSpaces[]  = "     ";

/* Searches and lists all files in the first directory
 * for the first CD-ROM device on the bus. */
```

Example 17-2. CD Search Example (cont.)

```
USI SearchCD(void)
{
ULI     Dh;
UC      *SearchData;
UC      *VersionInfo;
sbType  *FileSpec;
UC      rgbFileSpec[50];
USI     cbFileSpec;
USI     erc;
USI     ercClose;

   CheckErc (AllocMemorySL(4096, &SearchData));
   CheckErc (AllocMemorySL(100, &FileSpec));
   CheckErc (AllocMemorySL(100, &VersionInfo));

   /* Display the request we will execute */
   NewLine();
   OutputToVid0(&rgbVersionRequest,
            sizeof(rgbVersionRequest)-1);
   NewLine();
   NewLine();

   erc = CdVersion(&rgbDevName, sizeof(rgbDevName)- 1,
       VersionInfo, 20);
   if (erc == ercOK)  {
      cbDevName = *(VersionInfo+4);
      movb((VersionInfo+5), &rgbDevName[1], cbDevName);
      rgbDevName[0] = '[';
      rgbDevName[cbDevName+1] = '0';
      rgbDevName[cbDevName+2] = ']';
      cbDevName += 3;

      /* Display the device name */
      OutputToVid0(&rgbServiceName,
         sizeof(rgbServiceName)-1);
      movb((VersionInfo+4), &rgbDeviceName[0],
         *(VersionInfo+4)+1);
      OutputToVid0(&rgbDeviceName[1], rgbDeviceName[0]);
      NewLine();
```

Example 17-2. CD Search Example (cont.)

```
    /* Display the revision of the installed CD-ROM
     * Service */
    OutputToVid0(&rgbVersionLevel, sizeof
       (rgbVersionLevel)-1);
    rgbVersion[0] = *(VersionInfo+1)%10 + 0x30;
    rgbVersion[2] = *VersionInfo + 0x30;
    OutputToVid0(&rgbVersion, sizeof(rgbVersion)-1);
    NewLine();

    /* Display the hardware environment */
    OutputToVid0(&rgbHardware,sizeof(rgbHardware)-1);
    if (*(VersionInfo+2) == 1)
       OutputToVid0(&rgbNGen,sizeof(rgbNGen)-1);
    else
       OutputToVid0(&rgbSRP,sizeof(rgbSRP)-1);

    /* Display the number of CD-ROM drives present */
    NewLine();
    OutputToVid0(&rgbNumDrives,sizeof(rgbNumDrives)-1);
    rgbDriveCount[0] = *(VersionInfo+3) + 0x30;
    OutputToVid0(&rgbDriveCount,sizeof(rgbDriveCount)-
       1);
    NewLine()
}

/* Generate the full search file specification */
movb(&rgbDevName, &rgbFileSpec, cbDevName);
movb(rgbSearchSpec, &rgbFileSpec[cbDevName],
   sizeof(rgbSearchSpec)-1);
cbFileSpec = cbDevName + sizeof(rgbSearchSpec)-1;

/* Display the request being executed along with the
 * search file specification.
 */
NewLine();
OutputToVid0(&rgbSearchFirst, sizeof(rgbSearchFirst)- 1);
NewLine();
NewLine();
OutputToVid0(&rgbSearchFor, sizeof(rgbSearchFor)-1);
OutputToVid0(&rgbFileSpec, cbFileSpec);
NewLine();
```

Example 17-2. CD Search Example (cont.)

```
/* Set file spec to all white space */
setb(0x20, FileSpec, sizeof(sbType));

/* Search for first match */
erc = ercCDHeapFull;
while (erc == ercCDHeapFull)  {
   /* Find the first file that matches our spec */
   erc = CdSearchFirst(&rgbFileSpec, cbFileSpec, NIL,
      0, NIL, 0, FileSpec, 2048, &Dh);
   if (erc != ercOK && erc != ercCDHeapFull) return erc;
}

/* Time to display the file name */
NewLine();
OutputToVid0(&rgbFiveSpaces, sizeof(rgbFiveSpaces)-1);
OutputToVid0(&FileSpec->rg, FileSpec->cb);
NewLine();

/* Find all other files that match our spec */
NewLine();
OutputToVid0(&rgbSearchNext, sizeof(rgbSearchNext)-1);
NewLine();
NewLine();

/* Loop until all matches have been found */
erc = ercOK;
while (erc == ercOK)  {
   erc = CdSearchNext(Dh, NIL, 0, FileSpec, 2048);
   if (erc == ercOK)  {
      /* Time to display the file name */
      OutputToVid0(&rgbFiveSpaces,
         sizeof(rgbFiveSpaces)-1);
      OutputToVid0(&FileSpec->rg, FileSpec->cb);
                        NewLine();
   }
   else if (erc == ercCDHeapFull)
      erc = ercOK;
}
```

Example 17-2. CD Search Example (cont.)

```
    /* This is the erc we expect when all matches have
     * been found */
    if (erc == ercCDNoSuchFile)
       erc = ercOK;

    NewLine();
    OutputToVid0(&rgbSearchClose, sizeof(rgbSearchClose)-
       1);
    NewLine();

    /* Must perform a CdSearchClose to free-up the file
     * handle and the CDROM Service search buffers.
     */
    ercClose = CdSearchClose(Dh);
    if (erc == ercOK && ercClose != ercOK)
       erc = ercClose;
    return erc;
}
```

Copying a CD-ROM File to Disk

DiscThis example shows copying a CD-ROM file to disk. First, it uses CdGetVolumeInfo to determine whether the CD-ROM file is in ISO or High Sierra format. Then it uses the CdGetDirEntry operation to obtain the directory record of the source file, which tells how long the file is. Once you know the length of the file to be copied, you can create and open a disk file and write to it, as shown here.

Compile and link this example as described earlier for Example 1. Then use the **Run** command to invoke the program for Example 4 with the following parameters:

```
Run
 Run file          example4.run
 [Case]
 [Command]
 [Parameter 1]     [CDROM0]<DIR>FILE_NAME
 [Parameter 2]     [DISC VOLUME]<DIR>FILE_NAME
```

Example: Copying to Disk

Example 17-3. Copying a CD-ROM File to Disk

```
/*
 * Example CD-ROM API application.  This application takes
 * two input parameters, the name of the CD-ROM file you
 * wish to copy from and the name of the disc file you wish
 * to copy to.
 * The following illustrates the command form and gives an
 * example of how it might be used.  This is example3.
 */
/* CD Copy
 *   CD-ROM file to copy      [CDROM0]<Sys>readme.doc
 *   Destination              [f0]<sys>readme.doc
 * END OF MODULE HEADER
 */
```

Example 17-3. Copying a CD-ROM File to Disk (cont.)

```
#define  Syslit
#define  Sublit
#define  sdType
#define  FhbType
#define  CreateFile
#define  CdClose
#define  CdGetDirEntry
#define  CdGetVolumeInfo
#define  CdOpen
#define  CdRead
#define  CheckErc
#define  CloseFile
#define  GetFileStatus
#define  OpenFile
#define  RgParam
#define  SetFileStatus
#define  ULCmpB
#define  Write
#include <CtosLib.h>

#define  RxErc
#include <erc.h>
#include <stdlib.h>
#include <stdio.h>
#include "CdRom_Types.h"
   /* Defines request specific data structures */
#include "CdRom_Formats.h"
   /* Defines ISO and HSG data structures      */
#include "CdRomService.h"
   /* Defines service specific literals        */

typedef  unsigned short int   USI;    /* word */
typedef  unsigned char        UC;     /* byte */
typedef  unsigned long int    ULI;    /* dword */
typedef  void _far *          Pointer;
```

Example 17-3. Copying a CD-ROM File to Disk (cont.)

```
sdType              *From;
sdType              *To;
UC                  *CdBuff;
IsoDirRecordType    *IsoDirRec;
HsgDirRecordType    *HsgDirRec;
FhbType             *Fhb;
sdType              FileFrom;
sdType              FileTo;
FhbType             ToFhb;
UC                  fFormatType;
ULI                 dataLength;
ULI                 CDh;
USI                 fh;
ULI                 qFileSize;
char                rgbIsoIdentifier[] = "CD001";
char                rgbHsgIdentifier[] = "CDROM";

void main(void)
{
ULI   lfaCD;
USI   wDataRet;
USI   i;
USI   ercDisc;
USI   ercDisc;

   From = &FileFrom;
   To   = &FileTo;
   Fhb  = &ToFhb;

   CheckErc(RgParam (1, 0, From));
   CheckErc(RgParam (2, 0, To));
   CdBuff = (Byte *) malloc(2048);
   if (CdBuff == NULL)
      exit(0);
   /* Open the CD-ROM file and perform a CdGetVolumeInfo
    * request to determine the format of the target
    * CD-ROM disc.
    */
```

Example 17-3. Copying a CD-ROM File to Disk (cont.)

```
    CheckErc(CdOpen(0, From->pb, From->cb, NULL, 0,&CDh));
    CheckErc(CdGetVolumeInfo(CDh, CdBuff, 2048, &wDataRet,
    4));i = ULCmpB(CdBuff+1, &rgbIsoIdentifier, 5);

    if (i != 0xFFFF)  {
       i = ULCmpB(CdBuff+9, &rgbHsgIdentifier, 5);
       if (i != 0xFFFF)  {
          free(CdBuff);
          exit(0);
       }
       else
          fFormatType = HSG_DISC;
    }
    else
       fFormatType = ISO_DISC;

    /* Perform a CdGetDirEntry request to obtain the
     * directory entry of the source file so that we know
     * how large it is.
     */
    CheckErc(CdGetDirEntry(From->pb, From->cb, CdBuff,
             2048, &wDataRet));
    if (fFormatType == ISO_DISC)  {
       IsoDirRec = (IsoDirRecordType *) CdBuff;
       dataLength = IsoDirRec->sFileSectionLSBF;
    }
    else  {
       HsgDirRec = (HsgDirRecordType *) CdBuff;
       dataLength = HsgDirRec->sFileSectionLSBF;
    }
/* Create and open the disc file we will be writing to */
    qFileSize = dataLength;
    offsetof(qFileSize) = (offsetof(qFileSize) + 511) & (~
    511); CheckErc(CreateFile(To->pb, To->cb, NIL, 0,
         qFileSize));
    CheckErc(OpenFile(&fh, To->pb, To->cb, NIL, 0,
      modeModify));
    lfaCD   = 0;
    ercDisc = ercOK;
    ercDisc = ercOK;
```

Example 17-3. Copying a CD-ROM File to Disk (cont.)

```
    /* Read the CD-ROM file and write to the disc file
     * until we hit EOF
     */
    while(ercDisc == ercOK && ercDisc == ercOK)  {
       ercDisc = CdRead(CDh, lfaCD, CdBuff, 2048, &wDataRet);

       /*
        * When writing to a disc file the data length must
        * be a multiple of 512
        */
       if (wDataRet != 2048)
          wDataRet = (wDataRet + 511) & (~ 511);
       ercDisc = Write(fh, CdBuff, wDataRet, lfaCD,
           &wDataRet);
       lfaCD += wDataRet;
    }
    /* Update the FHB with the correct data length */
    CheckErc(GetFileStatus(fh, 12, Fhb, 512));
    Fhb->endOfFileLfa = dataLength;
    CheckErc(SetFileStatus(fh, 12, Fhb, 512));
    CheckErc(CloseFile(fh));

    CheckErc(CdClose(CDh));
    free(CdBuff);
    exit(0);
}
```

Using Audio Features of CD-ROM

If a disc does not contain data in High Sierra or ISO format, it may be an audio disc. Audio functions of the CD-ROM Service are handled by the CdAudioCtl operation.

Specifying Locations on the CD-ROM Disc

Locations on a CD-ROM disc can be specified in one of two ways: by track number or by minute-second-frame (MSF). Track numbers are simply integers that refer to each track on the disc. MSF format refers to the format used in the Red Book standard created by N. V. Philips and Sony Corporation. The address-mode parameter of the Audio Play function indicates which form of addressing is used.

In MSF format, minute ranges are from 0 to 70, second ranges from 0 to 59, and frame ranges from 0 to 75. There are 75 frames in one second. MSF addresses are usually relative to the physical beginning of the disc. The first two seconds of information on a disc are reserved for the table of contents. The usable area of a CD-ROM disc thus starts at 0:2:0 (0 minutes, 2 seconds, 0 frames).

MSF format would be required if you wanted to play only a specific portion of a track. In a multimedia environment where data and audio are recorded on the same disc, MSF format would also be used.

Q-Channel

The Q-channel is a status feature found on compact discs. It contains useful information for control and addressing. Use the Audio Q-Channel Info function to return the current track number, the relative MSF address to the start of that track, and the absolute MSF address (that is, from the start of the disc). The code example in this section illustrates use of this function.

Audio Example

This example uses the Audio Disc Info function of CdAudioCtl to find out what is on the disc (low track number, high track number, and starting address of the lead-out track, in MSF format). It then uses the Audio Track Info function to obtain the MSF address for each audio track on the disc. (For purposes of illustration, this example uses the MSF form of addressing, since it is more complex than use of track numbers.)

Each track on the disc is played (Audio Play), and the progress is monitored through use of the Audio Q-Channel and Audio Status functions. When Audio Q-Channel detects that the play operation has proceeded to a new track, a pause is performed (Audio Pause), and the new track is displayed on the screen. Audio Resume then resumes the playing of the audio disc. When Audio Status detects that the audio play has completed, the program exits.

Compile, link, and run this example as described earlier for Example 1.

Example 17-4. Audio Example

```
/* MODULE HEADER
 * INCLUDE FILE: examples3.c
 * MACHINE: B20
 * LANGUAGE: METAWARE C V1.0
 * OS: BTOS
 * DESCRIPTION: A CD-ROM example that illustrates the use
 * of the CdAudioCtl request.  This is example4.
 * END OF MODULE HEADER
 */
#define  Syslit
#define  Sublit
#define  AllocMemorySL
#define  CdAudioCtl
#define  CdClose
#define  CdOpen
#define  CheckErc
#define  DeallocMemorySL
#define  DecOut
#define  Delay
#define  HexOut
#define  HexQdOut
```

Example 17-4. Audio Example (cont.)

```
#define  NewLine
#define  zPrint
#include <CtosLib.h>
#include <stdlib.h>
#include "examples.h"
#include "example3.h"

/* Perform Audio Disc Info function of the CdAudioCtl
 * request to obtain the low, high, and leadout tracks of
 * the installed CD-ROM disc.
 */
USI disc_info (ULI CDh, paramType *Parameter, discInfoType
                    *DiscInfo)
{
USI   erc;

   NewLine();
   zPrint(&rgbDiscInfo);
   NewLine();

   Parameter->function = AUDIO_DISC_INFO;
   erc = CdAudioCtl(CDh, Parameter, 1, DiscInfo, 20);
   if (erc != ercOK)
      return(erc);
   NewLine();
   zPrint(&rgbLowTrack);
   DecOut(DiscInfo->lowTrack);

   NewLine();
   zPrint(&rgbHiTrack);
   DecOut(DiscInfo->highTrack);

   NewLine();
   zPrint(&rgbLeadOutTrack);
   HexQdOut(DiscInfo->leadOut);
   NewLine();
   return (ercOK);
}
```

Example 17-4. Audio Example (cont.)

```
/* Perform Audio Track Info function of the CdAudioCtl
 * request to obtain Red Book address for the beginning
 * of each audio track on the installed CD-ROM disc.
 */
USI track_info (ULI CDh, trackParamType *Params,
        discInfoType *DiscInfo, trackLogType *TrackLog)
{
USI   i;
USI   erc;

   NewLine();
   zPrint(&rgbTrackInfo);
   NewLine();

   Params->function = AUDIO_TRACK_INFO;
   for (i = DiscInfo->lowTrack; i < DiscInfo-
            >highTrack+2; i++)  {
      if (i == DiscInfo->highTrack+1)
         Params->track = LEADOUT;
      else
         Params->track = i;

      erc = CdAudioCtl(CDh, Params, 2, TrackLog, 6);
      if (erc != ercOK)
         return (erc);

      /* Don't display leadout track location */
      if (i < DiscInfo->highTrack+1)  {
         NewLine();
         zPrint(&rgbTrack);
         DecOut(i);

         if (i > 9)
            zPrint(&rgbOneSpace);
         else
            zPrint(&rgbTwoSpaces);
```

Example 17-4. Audio Example (cont.)

```
            zPrint(&rgbMSF);
            HexOut(TrackLog->min);
            zPrint(&rgbSlash);
            HexOut(TrackLog->sec);
            zPrint(&rgbSlash);
            HexOut(TrackLog->frame);
         }
         else
            TrackLog->frame--;

         TrackLog++;
      }
      NewLine();
      return (ercOK);
}

/* Perform Audio Play, Audio Status, Audio Pause, and Audio
 * Resume functions of the CdAudioCtl request.  The Audio
 * Play function is used to begin playing of the installed
 * CD-ROM disc and the Audio Status function is performed to
 * monitor the progress.  When Audio Status detects that the
 * play operation has proceeded to a new track, Audio Pause
 * is performed and the new track is displayed on the
 * screen.  Audio Resume is then performed to commence the
 * playing of the audio disc.
 */
USI play_disc (ULI CDh, playMSFType *Params, discInfoType
               *DiscInfo, trackLogType *TrackLog,
               statusType *AudioStatus)
{
USI          i;
USI          j;
USI          current_track;
qChannelType *QChannel;
trackLogType *Track;
USI          erc;

   NewLine();
   zPrint(&rgbAudioPlay);
   NewLine();
```

Example 17-4. Audio Example (cont.)

```
Params->function    = AUDIO_PLAY;
Params->addrMode    = MSF;
Params->wait        = 0;
Params->startMin    = TrackLog->min;
Params->startSec    = TrackLog->sec;
Params->startFrame = TrackLog->frame;
Track = TrackLog + DiscInfo->highTrack;
Params->endMin      = Track->min;
Params->endSec      = Track->sec;
Params->endFrame    = Track->frame;

/* If the disc does not contain audio tracks we will
 * get an erc 15712 */
erc = CdAudioCtl(CDh, Params, 10, NIL, 0);
if (erc == ercCDHardwareError) {
   NewLine();
   zPrint(&rgbNoAudio);
   return (ercOK);
}

/* Get initial status */
Params->function = AUDIO_STATUS;
erc = CdAudioCtl(CDh, Params, 1, AudioStatus, 8);
if (erc != ercOK)
   return erc;

/* Display current track */
NewLine();
current_track = 0;
zPrint(&rgbPlayingTrack);
DecOut(current_track+1);
Track = TrackLog + current_track + 1;
QChannel = (qChannelType *) AudioStatus;
```

Example 17-4. Audio Example (cont.)

```
while (AudioStatus->statusByte == PLAY_IN_PROGRESS)  {
   /* Get the q-channel info so we can check the
    * current track */
   Params->function = AUDIO_Q_CHANNEL;
   erc = CdAudioCtl(CDh, Params, 1, QChannel, 20);
   if (erc != ercOK) {
      zPrint(&rgbBlinkOff);
      return erc;
   }
   /* We have moved on to the next track; update the
    * display */
   if (QChannel->track > current_track+1) {
      /* Pause when we reach the end of the track */
      Params->function   = AUDIO_PAUSE;
      erc = CdAudioCtl(CDh, Params, 10, NIL, 0);
      if (erc != ercOK) {
         zPrint(&rgbBlinkOff);
         return erc;
      }

      if (current_track >= 9)
         zPrint(rgbBackTrack1);
      else
         zPrint(rgbBackTrack2);

      zPrint(rgbBackUp);
      zPrint(rgbPausing);
      Delay(20);
      zPrint(rgbBackUp);
      zPrint(rgbPlayingOf);
      DecOut(current_track+1);
      zPrint(rgbComplete);
      NewLine();

      /* Resume playing the next track */
      Params->function   = AUDIO_RESUME;
      erc = CdAudioCtl(CDh, Params, 10, NIL, 0);
```

Example 17-4. Audio Example (cont.)

```
            if (erc != ercOK) {
               zPrint(&rgbBlinkOff);
               return erc;
            }
            zPrint(rgbPlayingTrack);
            Track++;
            current_track++;
            DecOut(current_track+1);
         }

         /* Update the audio status */
         Params->function = AUDIO_STATUS;
         erc = CdAudioCtl(CDh, Params, 1, AudioStatus, 8);
         if (erc != ercOK) {
            zPrint(&rgbBlinkOff);
            return erc;
         }
      }
      /* Display completion of last track */
      if (AudioStatus->statusByte == PLAY_COMPLETE) {
         if (current_track >= 9)
            zPrint(rgbBackTrack1);
         else
            zPrint(rgbBackTrack2);

         zPrint(rgbBackUp);
         zPrint(rgbPlayingOf);
         DecOut(current_track+1);
         zPrint(rgbComplete);
         NewLine();
      }
      else
         zPrint(&rgbBlinkOff);
      return (ercOK);
}
```

Example 17-4. Audio Example (cont.)

```
void main (void)
{
trackLogType  *TrackLog;
discInfoType  *DiscInfo;
UC            *AudioParams;
UC            *AudioInfo;
ULI           CDh;

   CheckErc (AllocMemorySL(20,   &DiscInfo));
   CheckErc (AllocMemorySL(180, &TrackLog));
   CheckErc (AllocMemorySL(200, &AudioParams));
   CheckErc (AllocMemorySL(200, &AudioInfo));
   setb(0xFF, TrackLog, 180);

   CheckErc (CdOpen(modeCDShare, &rgbDevName,
     sizeof(rgbDevName)-1, NULL, 0, &CDh));
   CheckErc (disc_info(CDh, (paramType *) AudioParams,
             DiscInfo));
   CheckErc (track_info(CDh, (trackParamType *)
             AudioParams, DiscInfo,
                   TrackLog));
   CheckErc (play_disc(CDh, (playMSFType *) AudioParams,
             DiscInfo, TrackLog, (statusType *)
             AudioInfo));
   CheckErc (CdClose(CDh));
   CheckErc (DeallocMemorySL(AudioInfo, 200));
   CheckErc (DeallocMemorySL(AudioParams, 200));
   CheckErc (DeallocMemorySL(TrackLog, 180));
   CheckErc (DeallocMemorySL(DiscInfo, 6));
}
```

File Formats

This section lists ISO and High Sierra file formats for data that is returned by CD-ROM Service operations:

ISO Primary Volume Descriptor
High Sierra Primary Volume Descriptor

ISO Directory Record Format
High Sierra Directory Record Format

This section also describes the following character sets referred to in the volume descriptor structures:

d-characters
a-characters
c-characters
a1-characters
d1-characters
separators

Primary Volume Descriptor Formats

Table 17-1 lists the format for an ISO primary volume descriptor. Each field is described in more detail below the table.

Table 17-1. ISO Primary Volume Descriptor

Byte	Field	Content
1	*Volume descriptor type*	Numeric value
2-6	*Standard identifier*	CD001
7	*Volume descriptor version*	Numeric value
8	Unused field	(00) bytes
9-40	*System identifier*	a-characters
41-72	*Volume identifier*	d-characters
73-80	Unused field	(00) bytes

continued

Table 17-1. ISO Primary Volume Descriptor (cont.)

Byte	Field	Content
81-88	*Volume space size*	Numeric value
89-120	Unused field	(00) bytes
121-124	*Volume set size*	Numeric value
125-128	*Volume sequence number*	Numeric value SEPARATOR 1, SEPARATOR 2
129-132	*Logical block size*	Numeric value
133-140	*Path table size*	Numeric value
141-144	*Location of occurrence of type L path table*	Numeric value
145-148	*Location of optional occurrence of type L path table*	Numeric value
149-152	*Location of occurrence of type M path table*	Numeric value
153-156	*Location of optional occurrence of type M path table*	Numeric value
157-190	*Directory record for root directory*	34 bytes
191-318	*Volume set identifier*	d-characters
319-446	*Publisher identifier*	a-characters
447-574	*Data preparer identifier*	a-characters
575-702	*Application identifier*	a-characters
703-739	*Copyright file identifier*	d-characters,
740-776	*Abstract file identifier*	d-characters, SEPARATOR 1, SEPARATOR 2
777-813	*Bibliographic file identifier*	d-characters, SEPARATOR 1, SEPARATOR 2

continued

Table 17-1. ISO Primary Volume Descriptor (cont.)

Byte	Field	Content
814-830	*Volume creation date and time*	Digit(s), numeric value
831-847	*Volume modification date and time*	Digit(s), numeric value
848-864	*Volume expiration date and time*	Digit(s), numeric value
865-881	*Volume effective date and time*	Digit(s), numeric value
882	*File structure version*	Numeric value
883	Reserved for future standardization	(00) byte
884-1395	*Application use*	Not specified
1396-2048	Reserved for future standardization	(00) byte

Volume descriptor type

A 1 in this field indicates that the volume descriptor is a standard file structure volume descriptor. (8 bits)

Standard identifier

Specifies the standard to which the volume descriptor is expected to conform. CD001 indicates the ISO standard.

Volume descriptor version

Specifies the version of the volume structure standard to which the volume descriptor is expected to conform. A 1 indicates the present standard. (8 bits)

Unused field

Is set to (00).

System identifier

Specifies an identification of a system that can recognize and act on the content of the Logical Sectors with Logical Sector Numbers 0 to 15 of the volume. See "Character Sets," later in this section for a list of a-characters.

Volume identifier

Identifies the volume. See "Character Sets," later in this section, for a list of d-characters.

Unused field

Is set to (00).

Volume space size

specifies the number of logical blocks in which the volume space is recorded. (32 bits)

Unused field

Is set to (00).

Volume set size

Specifies the number of volumes in the volume set of which the volume is a member. (16 bits)

Volume sequence number

Specifies the ordinal number of the volume in the volume set of which the volume is a member. (16 bits)

Logical block size

Specifies the size, in bytes, of a logical block. (16 bits)

Path table size

Specifies the length, in bytes, of a recorded occurrence of the path table identified by this volume descriptor. (32 bits)

Location of occurrence of type L path table

Specifies the logical block number (LBN) of the first logical block allocated to the extent that contains an occurrence of the path table. Multibyte numeric values in a record of this occurrence of the path table are recorded with the least significant byte first.

Location of optional occurrence of type L path table

Specifies the LBN of the first logical block allocated to the extent that contains an optional occurrence of the path table. A value of 0 indicates that the extent is not expected to have been recorded. Multibyte numeric values in a record of this occurrence of the path table are recorded with the least significant byte first. (32 bits)

Location of occurrence of type M path table

Specifies the LBN of the first logical block allocated to the extent that contains an occurrence of the path table. Multibyte numeric values in a record of this occurrence of the path table are recorded with the most significant byte first. (32 bits)

Location of optional occurrence of type M path table

Specifies the logical block number of the first logical block allocated to the extent that contains an optional occurrence of the path table. A value of 0 indicates that the extent is not expected to have been recorded. Multibyte numeric values in a record of this occurrence of the path table are recorded with the most significant byte first.

Directory record for root directory

Contains the directory record for the root directory.

Volume set identifier

Specifies an identification of the volume set of which the volume is a member. See "Character Sets," later in this section for a list of d-characters.

Publisher identifier

Specifies an identification of the user who specified what to record on the volume group of which the volume is a member.

If the first byte is set to (5F), the remaining bytes of this field specify an identifier for a file containing the identification of the user. This file is described in the root directory. The file name may not contain more than eight d-characters, and the file name extension may not contain more than three d-characters.

Setting all bytes in this field to (20) indicates that no such user is identified. See "Character Sets," later in this section for a list of a-characters.

Data preparer identifier

Specifies an identification of the person or other entity that controls the preparation of the data to be recorded on the volume group of which the volume is a member.

If the first byte is set to (5F), the remaining bytes of this field specify an identifier for a file containing the identification of the data preparer. This file is described in the root directory. The file name may not contain more than eight d-characters, and the file name extension may not contain more than three d-characters.

Setting all bytes in this field to (20) indicates that no such data preparer is identified. See "Character Sets," later in this section for a list of a-characters.

Application identifier

Specifies an identification of the specification of how the data are recorded on the volume group of which the volume is a member.

If the first byte is set to (5F), the remaining bytes of this field specify an identifier for a file containing the identification of the application. This file is described in the root directory. The file name may not contain more than eight d-characters, and the file name extension may not contain more than three d-characters.

Setting all bytes in this field to (20) indicates that no such application is identified. See "Character Sets," later in this section for a list of a-characters.

Copyright file identifier

Specifies an identification for a file described by the root directory and containing a copyright statement for those volumes of the volume set, whose sequence numbers are less than or equal to the assigned volume set size of the volume. Setting all bytes in this field to (20) indicates that no such file is identified.

The file name of a copyright file may contain no more than eight d-characters. The file name extension of a copyright file identifier may not contain more than three d-characters.

See "Character Sets," later in this section for a list of d-characters.

Abstract file identifier

Specifies an identification for a file described by the root directory containing an abstract statement for those volumes of the volume set whose sequence numbers are less than or equal to the assigned volume set size of the volume. Setting all bytes in this field to (20) indicates that no such file is identified.

The file name of an abstract file may contain no more than eight d-characters. The file name extension of an abstract file identifier may not contain more than three d-characters. See "Character Sets," later in this section for a list of d-characters.

Bibliographic file identifier

Specifies an identification for a file described by the root directory and containing bibliographic records interpreted according to standards that are the subject of an agreement between the originator and the recipient of the volume. Setting all bytes in this field to (20) indicates that no such file is identified.

The file name of a bibliographic file identifier may contain no more than eight d-characters. The file name extension of a bibliographic file identifier may contain no more than three d-characters. See "Character Sets," later in this section for a list of d-characters.

Volume creation date and time

Specifies the date and time of day at which the information in the volume was created. The date and time are represented by a 17-byte field as described in the table below. If all the characters in bytes 1 through 16 are the digit ZERO and the number in byte 17 is 0, the date and time are not specified. Each byte contains the following information:

Byte	Field	Content
1-4	Year from 1 to 9999	Digits
5-6	Month from 1 to 12	Digits
7-8	Day from 1 to 31	Digits
9-10	Hour from 0 to 23	Digits
11-12	Minute from 0 to 59	Digits
13-14	Second from 0 to 59	Digits
15-16	Hundredths of second	Digits
17	Offset from Greenwich Mean Time in number of 15-minute intervals from -48 West to +52 East	Numeric value

Volume modification date and time

Specifies the date and time of day at which the information in the volume was last modified. (See the format for date and time, under *Volume creation date and time* above.)

Volume expiration date and time

Specifies the date and time of day at which the information in the volume can be regarded as obsolete. If the date and time are not specified, then the information is not considered obsolete. (See the format for date and time, under *Volume creation date and time* above.)

Volume effective date and time

Specifies the date and time of day at which the information in the volume can be used. If the date and time are not specified, the information can be used immediately. (See the format for date and time, under *Volume creation date and time* above.)

File structure version

Specifies the version of the specification of the records of a directory and of a path table. A value of 1 indicates the structure of this International Standard.

Reserved for future standardization

Is set to (00).

Application use

Is reserved for application use. Its content is not specified by this standard.

Reserved for future standardization

Is set to (00).

Table 17-2 lists the format for a High Sierra primary volume descriptor. Each field is described in more detail below the table.

Table 17-2. High Sierra Primary Volume Descriptor

Byte	Field	Content
1-8	*Volume descriptor LBN*	Numeric value
9	*Volume descriptor type*	Numeric value
10-14	*Volume structure standard identifier*	CDROM
15	*Volume structure standard version*	Numeric value
16	Reserved	(00) byte
17-48	*System identifier*	a-characters
49-80	*Volume identifier*	d-characters
81-88	Reserved	(00) bytes
89-96	*Volume space size*	Numeric value
97-128	Reserved	(00) bytes
129-132	*Volume set size*	Numeric value
133-136	*Volume set sequence number*	Numeric value
137-140	*Logical block size (LBS)*	Numeric value
141-148	*Path table size*	Numeric value
149-152	*Location of first mandatory occurrence of path table*	Numeric value
153-156	*Location of optionall occurrence of path table*	Numeric value
157-160	*Location of optional occurrence of path table*	Numeric value
161-164	*Location of optional occurrence of path table*	Numeric value

continued

Table 17-2. High Sierra Primary Volume Descriptor (cont.)

Byte	Field	Content
165-168	*Location of second mandatory occurrence of path table*	Numeric value
169-172	*Location of optional occurrence of path table*	Numeric value
173-176	*Location of optional occurrence of path table*	Numeric value
177-180	*Location of optional occurrence of path table*	Numeric value
181-214	*Directory record for root directory*	34 bytes
215-342	*Volume set identifier*	d-characters
343-470	*Publisher identifier*	a-characters
471-598	*Data preparer identifier*	a-characters
599-726	*Application identifier*	a-characters
727-758	*Copyright file identifier*	d-characters, FULL STOP
759-790	*Abstract file identifier*	d-characters, FULL STOP
791-806	*Volume creation date and time*	Digit(s)
807-822	*Volume modification date and time*	Digit(s)
823-838	*Volume expiration date and time*	Digit(s)
839-854	*Volume effective date and time*	Digit(s)
855	*File structure standard version*	Numeric value
856	Reserved	(00) byte
857-1368	*Application use*	Not specified
1369-2048	Reserved	(00) bytes

Volume descriptor LBN

Specifies the logical block number (LBN) of the first logical block allocated to the primary volume descriptor. (32 bits)

Volume descriptor type

A 1 in the field indicates that the volume descriptor is a standard file structure volume descriptor. (8 bits)

Volume structure standard identifier

Specifies the standard to which the volume descriptor is expected to conform. CDROM indicates the High Sierra standard.

Volume structure standard version

Specifies the version of the volume structure standard to which the volume descriptor is expected to conform. A 1 indicates the present standard. (8 bits)

Reserved

Is set to (00).

System identifier

Specifies an identification of a system that can recognize and act on the content of the Logical Sectors with Logical Sector Numbers 0 to 15 of the volume. See "Character Sets," later in this section for a list of a-characters.

Volume identifier

Identifies the volume. See "Character Sets," later in this section, for a list of d-characters.

Reserved

Is set to (00).

Volume space size

Specifies the number of logical blocks in which the volume space is recorded. (32 bits)

Reserved

Is set to (00).

Volume set size

Specifies the number of volumes in the volume set of which the volume is a member. (16 bits)

Volume set sequence number

Specifies the ordinal number of the volume in the volume set of which the volume is a member. (16 bits)

Logical block size (LBS)

Specifies the size, in bytes, of a logical block. (16 bits)

Path table size

Specifies the length, in bytes, of the path table. (32 bits)

Location of first mandatory occurrence of path table

Specifies the logical block number (LBN) of the first logical block allocated to the extent that contains the first mandatory occurrence of the path table. Multibyte number values in a record of this occurrence of the path table are recorded with the least significant byte first. (32 bits)

Location of optional occurrence of path table

Specifies the LBN of the first logical block allocated to the extent that contains an optional occurrence of the path table. A value of 0 indicates that the extent is not expected to have been recorded. Multibyte numeric values in a record of this occurrence of the path table are recorded with the least significant byte first. (32 bits)

Location of second mandatory occurrence of path table

Specifies the LBN of the first logical block allocated to the extent that contains the second mandatory occurrence of the path table. Multibyte numeric values in a record of this occurrence of the path table are recorded with the most significant byte first. (32 bits)

Location of optional occurrence of path table

Specifies the LBN of the first logical block allocated to the extent that contains an optional occurrence of the path table. A value of 0 indicates that the extent is not expected to have been recorded. Multibyte numeric values in a record of this occurrence of the path table are recorded with the most significant byte first. (32 bits)

Directory record for root directory

Contains the directory record for the root directory.

Volume set identifier

Specifies an identification of the volume set of which the volume is a member. See "Character Sets," later in this section, for a list of d-characters.

Publisher identifier

Specifies an identification of the user who specified what to record on the volume. See "Character Sets," later in this section, for a list of a-characters.

Data preparer identifier

Specifies an identification of the person or other entity that controls the preparation of the data recorded on the volume. See "Character Sets," later in this section, for a list of a-characters.

Application identifier

Specifies an identification of the specification for how the data are recorded on the volume. See "Character Sets," later in this section, for a list of a-characters.

Copyright file identifier

Specifies an identification for a file described by the root directory containing a copyright statement for the volume. A value in this field of all SPACEs indicates that the file is not expected to have been recorded.

The file name of a copyright file identifier is limited to 8 d-characters. The file name extension of a copyright file identifier is limited to 3 d-characters. See "Character Sets" later in this section, for a list of d-characters.

Abstract file identifier

Specifies an identification for a file described by the root directory containing an abstract statement for the volume. A value in this field of all SPACEs indicates that the file is not expected to have been recorded.

The file name of an abstract file identifier is limited to 8 d-characters. The file name extension of an abstract file identifier is limited to 3 d-characters.

Volume creation date and time

Specifies the date and time of day at which the information in the volume was created. The date and time are represented by a 16-byte field as described in the table below. If all characters of this field are the digit ZERO, the date and time are not specified. Each byte contains the following information:

Byte	Field	Content
1-4	Year from 1 to 9999	Digits
5-6	Month of the year from 1 to 12	Digits
7-8	Day of the month from 1 to 31	Digits
9-10	Hour of the day from 0 to 23	Digits
11-12	Minute of the hour from 0 to 59	Digits
13-14	Second of the minute from 0 to 59	Digits
15-16	Hundredths of a second	Digits

Volume modification date and time

Specifies the date and time of day at which the information in the volume was last modified. (See the format for date and time, under *Volume creation date and time* above.)

Volume expiration date and time

Specifies the date and time of day at which the information in the volume can be regarded as obsolete. If the date and time are not specified, then the information is not considered obsolete. (See the format for date and time, under *Volume creation date and time* above.)

Volume effective date and time

Specifies the date and time of day at which the information in the volume can be used. If the date and time are not specified, then the information can be used immediately. (See the format for date and time, under *Volume creation date and time* above.)

File structure standard version

Specifies the version of the file structure standard to which all directory records and all path tables are expected to conform. A value of 1 indicates the present standard. (8 bits)

Reserved

Is set to (00).

Application use

Is reserved for application use. Its content is not specified by this standard.

Reserved for future standardization

Is set to (00).

ISO Directory Record Format

The CdGetDirEntry request returns the directory record for the specified path. The format for an ISO directory record is described in Table 17-3. The fields are described in more detail following the table.

Table 17-3. ISO Directory Record Format

Off-set	Field	Size (Bytes)	Description
0	*Length of directory record*	0	Length of the ISO directory record
1	*Extended attribute record length*	1	Number of logical blocks recorded
2	*Location of extent LSBF*	4	LBN of first logical block in extent
6	*Location of extent MSBF*	4	LBN of first logical block in extent
10	*Data length LSBF*	4	Size of the file section
14	*Data length MSBF*	4	Size of the file section
18	*Recording date and time*	7	Date and time of recording
25	*File flags*	1	Flags for file options
26	*File unit size*	1	File unit size for file section
27	*Interleave gap size*	1	Number of consecutive logical blocks
28	*Volume sequence number*	4	Ordinal number of volume in volume set
32	*Length of file identifier*	1	Length in bytes of *File identifier* field
33	*File identifier*	?	Identification for file or directory (variable size)
	Padding field	1	Unused value

Length of directory record

Is the length, in bytes, of the ISO directory record. The maximum length of a directory record is 68 bytes (33-byte record header + 34-byte maximum file identifier + 1-byte padding field).

Extended attribute record length

Is the number of logical blocks that make up the extended attribute record preceding the file data in the extent. If no extent is recorded, this value is 0. The maximum length of an extended attribute record is 65,786 bytes (251-byte record header + 65,535-byte application use field + 0-byte escape sequence field).

Location of extent LSBF

Is the logical block number of the first logical block allocated to the extent, with least significant bit first (for Intel processors, for example).

Location of extent MSBF

Is the logical block number of the first logical block allocated to the extent, with most significant bit first (for Motorola processors, for example).

Data length LSBF

Is the length, in bytes, of the file section, with least significant bit first.

Data length MSBF

Is the length, in bytes, of the file section, with most significant bit first.

Recording date and time

Are the date and time the recording was made. Each byte contains the following information:

Byte Number	Description
0	Number of years since 1990.
1	Month of the year, from 1 to 12.
2	Day of the month, from 1 to 31.
3	Hour of the day, from 0 to 23.
4	Minute of the hour, from 0 to 59.
5	Second of the minute, from 0 to 59.
6	Offset from Greenwich Mean Time, in 15 minute intervals from -48 (West) to +52 (East). This value is represented in binary notation by an 8-bit two's complement number.

File flags

Is a series of bit flags. Each bit, when set to the appropriate state, indicates the following:

Bit Number and Name	Bit State	Description
0 (Existence)	0	Upon inquiry, the user is informed of the file's existence.
	1	The existence of the file need not be made known to the user.
1 (Directory)	0	The directory record does not identify a directory.
	1	The directory record identifies a directory.
2 (Assoc. file)	0	The file is not an associate file.
	1	The file is an associate file.
3 (Record)	0	The structure of the information in the file is not specified by the Record Format field of any associated extended attribute record.
	1	The structure of the information in the file has a record format specified by a number other than 0 in the Record Format field of the extended attribute record.
4 (Protection)	0	An owner identification and a group identification are not specified for the file. Any user may read or execute the file.
	1	An owner identification and a group identification are specified for the file. At least one of the even-numbered bits or bit 0 in the Permissions field of the associated extended attribute record is set to 1.
5		Reserved; set to 0.
6		Reserved; set to 0.
7 (Multi-extent)	0	This directory record is the final directory record for the file.
	1	This is not the final directory record for the file.

File unit size

Is the file unit size of the file section, provided the file section is recorded in interleave mode. Otherwise, this value is 0.

Interleave gap size

Is the interleave gap size for the file section, provided the file section is recorded in interleave mode. Otherwise this value is 0.

Volume sequence number

Is the ordinal number of the volume in the volume set on which the extent described by this directory record is recorded.

Length of file identifier

Is the length in bytes of the *File identifier* field of the directory record.

File identifier

Is the identifier for a file or directory. If the directory bit (bit 1) of the *File flags* field is set to 0, the *File identifier* field is the identifier for a file. The characters in this field are d-characters or d1-characters, SEPARATOR 1, SEPARATOR 2.

If the Directory bit (bit 1) of the *File flags* field is set to 1, the File Identifier field is the identifier for a directory. The characters in this field are d-characters or d1-characters, or only a (00) byte, or only a (01) byte.

Padding field

Is an unused value that is present only if an even number is present in the *Length of file identifier* field.

The format for a High Sierra directory record is described in Table 17-4. The fields are described in more detail following the table.

Table 17-4. High Sierra Directory Record Format

Off-set	Field	Size (Bytes)	Description
0	*Length of directory record*	0	Length of the High Sierra directory record
1	*Extended attribute record length*	1	Number of logical blocks recorded
2	*Location of extent*	8	LBN of first logical block in extent
10	*Data length*	8	The size of the file section
18	*Recording date and time*	6	Date and time of recording
24	*File flags*	1	Flags for file options
25	Reserved	1	Reserved
26	*Interleave size*	1	number of logical blocks in extent
27	*Interleave skip factor*	1	number of logical blocks allocated to files separating of each part the file recorded in the extent
28	*Volume sequence number*	4	ordinal number of volume in volume set
32	*Length of file identifier*	1	length in bytes of *File Identifier* field
33	*File identifier*	?	identification for file or directory (variable size)
	Padding field	1	unused value

Length of directory record

Is the length, in bytes, of the High Sierra directory record. The maximum length of a directory record is 68 bytes (33-byte record header + 34-byte maximum file identifier + 1-byte padding field).

Extended attribute record length

Is the number of logical blocks that make up the extended attribute record preceding the file data in the extent. If no extent is recorded, this value is 0. The maximum length of an extended attribute record is 65,854 bytes (251-byte record header + 65,535-byte application use field + 68-byte directory record).

Location of extent

Is the logical block number of the first logical block allocated to the extent.

Data length

Is the length, in bytes, of the file section.

Recording date and time

Are the date and time the recording was made. Each byte contains the following information:

Byte Number	Description
0	Number of years since 1990
1	Month of the year, from 1 to 12
2	Day of the month, from 1 to 31
3	Hour of the day, from 0 to 23
4	Minute of the hour, from 0 to 59
5	Second of the minute, from 0 to 59

File flags

Is a series of bit flags. Each bit, when set to the appropriate state, indicates the following:

Bit Number and Name	Bit State	Description
0 (Existence)	0	Upon inquiry, the user is informed of the file's existence.
	1	The existence of the file need not be made known to the user.
1 (Directory)	0	The directory record does not identify a directory.
	1	The directory record identifies a directory, and the record bit will contain the value 0
2 (Assoc. file)	0	The file is not an associate file.
	1	The file is an associate file that will be ignored in interchange. The content will be interpreted differently by different systems.
3 (Record)	0	The structure of the information in the file is not specified by the *Record format* field of any associated extended attribute record.
	1	The structure of the information in the file has a record format specified by a number other than 0 in the *Record format* field of the extended attribute record.
4 (Protection)	0	The *Owner identification* and *Group identification* fields in the associated extended attribute record shall contain all zeros. Also, the *Permissions* field in the associated extended attribute record shall contain (AAAAh).
	1	The *Owner identification* and *Group identification* fields in the associated extended attribute record shall not contain all zeros. Also, the *Permissions* field in the associated extended attribute record shall contain values specified by the High Sierra Proposal.
5		Reserved; set to 0.
6		Reserved; set to 0.
7 (Multi-extent)	0	This directory record is the final directory record for the file.
	1	This directory record is not the final directory record for the file.

Reserved

Is set to (00).

Interleave size

Is the number of consecutive logical blocks in which each part of the file is recorded in the extent described by the directory record.

Interleave skip factor

Is the number of consecutive logical blocks allocated to other files separating each part of the file recorded in the extent described by the directory record. The number in this field is 0 if the directory bit (bit 1) of the *File flags* field is 1.

Volume sequence number

Is the ordinal number of the volume in the volume set on which the extent described by this directory record is recorded.

Length of file identifier

Is the length in bytes of the *File identifier* field of the directory record.

File identifier

Is the identifier for a file or directory. If the directory bit (bit 1) of the *File flags* field is set to 0, the *File identifier* field is the identifier for a file. The characters in this field are d-characters, FULL STOP, and SEMICOLON.

If the directory bit (bit 1) of the *File flags* field is set to 1, the *File identifier* field is the identifier for a directory. The characters in this field are d-characters, or only a (00) byte, or only a (01) byte.

Padding field

Is an unused value that is present only if an even number is present in the *Length of file identifier* field.

Character Sets

The following sections list the characters classified as d-characters, a-characters, c-characters, a1-characters, d1-characters, and separators.

d-characters

The following 37 characters are referred to as d-characters:

Character	Graphic	Code (hex)
Digits ZERO to NINE	0 ..	30 through 39
Capital letters A to Z	A .. Z	41 through 5A
Low line	_	5F

a-characters

The following 57 characters are referred to as a-characters:

Character	Graphic	Code (hex)
SPACE		20
EXCLAMATION MARK	!	21
QUOTATION MARK	"	22
PERCENT SIGN	%	25
AMPERSAND	&	26
APOSTROPHE	'	27
LEFT PARENTHESIS	(	28
RIGHT PARENTHESIS	)	29
ASTERISK	*	2A
PLUS SIGN	+	2B
COMMA	,	2C
HYPHEN	-	2D
PERIOD or FULL STOP	.	2E
SOLIDUS	/	2F
DIGITS ZERO to NINE	0 .. 9	30 through 39
COLON	:	3A
SEMICOLON	;	3B
LESS-THAN SIGN	<	3C
EQUALS SIGN	=	3D
GREATER-THAN SIGN	>	3E
QUESTION MARK	?	3F
CAPITAL LETTERS A to Z	A .. Z	41 through 5A
LOW LINE	_	5F

c-characters

The characters of the coded graphic character sets identified by the escape sequences in a supplementary volume descriptor are referred to as c-characters. This character set is used only in the ISO-9660 environment.

a1-characters

A subset of the c-characters is referred to as a1-characters. This subset is subject to agreement between the originator and the recipient of the volume. This character set is used only in the ISO-9660 environment.

d1-characters

A subset of the a1-characters is referred to as d1-characters. This subset is subject to agreement between the originator and the recipient of the volume. This character set is used only in the ISO-9660 environment.

Separators

The characters separating the components of a file identifier are

SEPARATOR 1	represented by the bit combination 2E(hex) - FULL STOP
SEPARATOR 2	represented by the bit combination 3B(hex) - SEMICOLON

Section 18
Sequential Access Service

Overview

The Sequential Access Service provides for transfer of data to and from a specified medium, at the proper position on the medium. It mediates the use of sequential access devices among a number of users. A device is opened for exclusive use by a single user and can be used by a different user only after it has been closed by the first user.

The Sequential Access Service is device-independent. Some of the devices currently supported are

- Quarter-inch cartridge (QIC) tape (either QIC or SCSI interfaces)
- Half-inch reel-to-reel tape (either shared resource processor Storage Processor or SCSI interfaces)
- 4mm digital data storage (DDS)

The Sequential Access Service must be installed on the workstation or SRP with the sequential access device. The workstation can be either a cluster server workstation or a cluster client workstation. If installed on the cluster server, client workstations of the server can also use the Sequential Access Service. Multiple instances of Sequential Access Services can be installed on an SRP, one on each processor board that controls a device. (For detailed information, see the *CTOS Executive Reference Manual.*)

The service provides both implicit and explicit routing to sequential access devices: if the device name is unique in the cluster, the request is routed to the named device. If two devices have the same name, the request is routed locally first, unless you specify the device at the server with an exclamation mark (!). Of course, performance can be affected by the relative placement of the service and the device. For example, if a volume archive program is running on a client workstation, performance is slower if the target tape drive is located on the server workstation than it would be if the target tape drive were located on the client workstation.

To use the operations described in this section, first use the **Install Sequential Access Service** command to install the Sequential Access Service on your workstation, as described in the *CTOS Executive Reference Manual* and the *CTOS System Administration Guide.*

Functional Groups of Operations

The following sections offer a brief description of the Sequential Access Service operations. See the *CTOS Procedural Interface Reference Manual* for complete descriptions of these operations.

Basic Operations

SeqAccessClose

Releases a sequential access device from the exclusive access rights granted by a previous call to SeqAccessOpen.

SeqAccessControl

Is used to specify certain medium positioning operations that do not involve the transfer of user data to the medium. These operations include rewinding, unloading, retensioning, erasing, and writing filemarks and may apply only to certain devices.

SeqAccessOpen

Provides exclusive access to the specified sequential access device and returns a sequential access handle to be used in subsequent operations.

SeqAccessRead

Reads data from the sequential access device and places it in a user-specified buffer.

SeqAccessStatus

Returns the current status of the sequential access device and its medium.

SeqAccessWrite

Writes data to the medium mounted on a sequential access device. This operation also verifies the data transfer and returns the amount of data unsuccessfully transferred.

Advanced Operations

SeqAccessCheckpoint

Causes the user to wait until all of the data supplied by previous SeqAccessWrite calls has been successfully transferred to the medium or until an exception condition occurs that prevents the transfer of data.

SeqAccessDiscardBufferData

Allows the user to discard buffered data from the output data stream. This operation is commonly used when an exception condition occurs (for example, end of medium, which signals that a new tape is to be mounted). It subsequently allows new operations, such as Write Filemark, to take place without transferring the previously buffered data to the medium.

SeqAccessModeQuery

Returns information about the current operating characteristics of the sequential access device, such as whether the device is write-protected, whether it is operating in buffered mode, and what recording density is being used for the medium.

SeqAccessModeSet

Configures operating characteristics of the sequential access device, such as buffered or unbuffered mode, medium transport speed, and medium recording density.

SeqAccessRecoverBufferData

Transfers data from the buffers of the Sequential Access Service to a user-specified buffer. This operation is used when an exception condition occurs that leaves data in the Sequential Access Service buffers. In this case, an application program may wish to recover this data before resuming write operations to the medium.

Miscellaneous

SeqAccessVersion

Returns version information for the specified sequential access device. This operation also returns the names of all the devices under control of the Sequential Access Service that is responsible for this device.

General Model of Sequential Access Devices

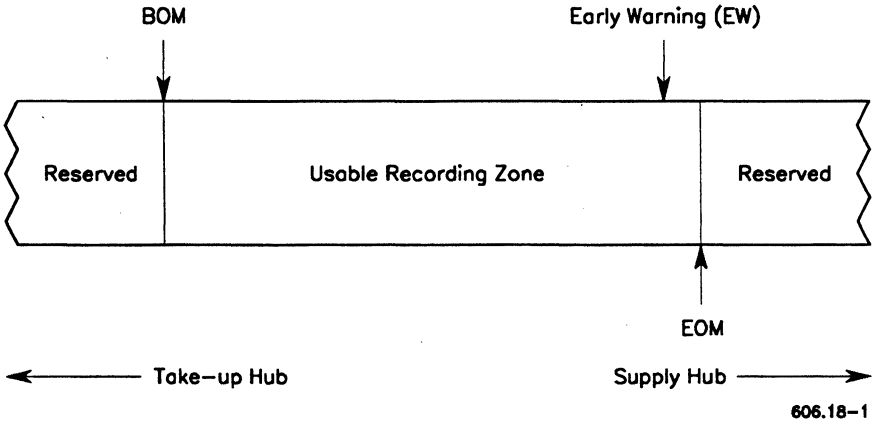
This section provides an overview of the data storage characteristics of sequential access devices, the logical elements within a tape, and how data is buffered on certain devices. These devices are used for the storage and retrieval of user data in a sequential manner, moving from beginning to end of the medium. For the purposes of this discussion, sequential access devices are generally tape devices, but they could be other device types as well. Position changes for sequential access devices typically take a long time, in contrast to position changes for direct access devices, such as disks, which are relatively rapid.

Data Storage Characteristics

The recording medium for tape devices consists of magnetic tape that is wound onto reels and may be enclosed in a cartridge that contains both the supply reel and the takeup reel. This recording medium has two physical attributes: beginning-of-medium (BOM) and end-of-medium (EOM). On most tapes, there are reserved areas at the beginning and end of the tape that are not used for recording, as shown in Figure 18-1.

Before EOM, there is also an area known as early warning (EW) that is reported to the program in time to recover unwritten data in the device's buffers and in the service's buffers before changing to a new tape and writing the remainder of the data. The amount of room left on the tape when early warning is encountered varies greatly for different devices. On half-inch tape devices, for example, recording densities range from 800 bits per inch to 6250 bits per inch. (On such devices, you can either determine the amount remaining empirically, or check the manufacturer's specifications for the device.) In addition, packing density (determined by block size and inter-block gaps; see below) also affects the amount of space left on the medium after early warning.

Figure 18-1. General Layout of a Tape

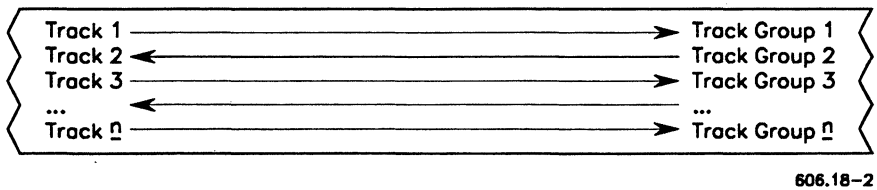


A track is the position on the medium where one write component records data. A device may read from or write to one or more tracks at a time.

The following paragraphs briefly describe how data is recorded on different types of devices. This information is provided as general background information, since this difference is transparent to the programmer.

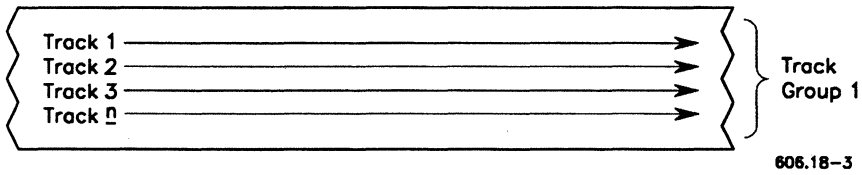
A QIC tape device reverses direction when it approaches the end-of-medium and continues writing in the opposite direction on track 2, as shown in Figure 18-2. When it reaches the beginning-of-medium on track 2, it again reverses direction and writes toward the end-of-medium on track 3. This type of recording is called serpentine recording.

Figure 18-2. Serpentine Recording (QIC Tape)



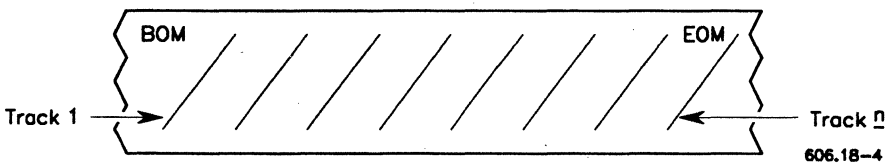
A half-inch tape device records all the tracks on the medium in parallel, moving once from beginning-of-medium to end-of-medium, as shown in Figure 18-3.

Figure 18-3. Parallel Recording (Half-inch Tape)



A DDS device records tracks diagonally across the medium, as shown in Figure 18-4. This type of recording is called helical scan.

Figure 18-4. Helical Scan Recording (DDS)



Logical Elements within a Tape

Within a tape, data is composed of collections of logical data blocks, filemarks, and blank space.

Logical Data Blocks

A logical data block is a unit of data supplied or requested by a user program. Logical blocks are stored according to the specifications of the format for the device and may be recorded as one or more physical blocks on the medium.

Interblock Gaps

Whenever a block or filemark is written, a gap is introduced on the medium between the block or filemark and whatever follows. With some storage formats such as half-inch tape, there is a sizable space between physical blocks (sometimes as much as three-quarters of an inch). In this case, increasing the size of the blocks speeds up the data transfer rate and increases the capacity of the tape. The tradeoff in this case is that larger blocks require larger extents of tape without defects.

Filemarks

Logical elements within a tape can be separated into discrete units by filemarks. Filemarks are special recorded elements containing no user data. They are often used to separate a group of data with a common origin or destination from another group of data with a common origin or destination. The SequentialAccessControl operation is used to write a filemark.

Blank Space

Blank space is generally found only at the end of the used portion of the medium. With some recording formats, end-of-data (EOD) is indicated by blank space on the medium. Other formats may use a convention (for example, two filemarks in a row) to indicate that there is nothing else on the tape.

Erase Gaps

In addition to blocks and filemarks, erase gaps can also be recorded on the medium (using the SequentialAccessControl operation), for error recovery purposes. For example, if a stretch of tape has medium defects that are making it impossible to write the data accurately, the Sequential Access Service can erase that section of tape and continue to an unharmed section of the medium.

Data Buffering

To improve throughput, the Sequential Access Service and many sequential access devices have buffers that act as a pipeline for data. At one end of this pipeline, you supply the data to be written. At the other end, that data is transferred physically from the pipeline onto the medium. (If you are reading from the device, the flow of data through the pipeline naturally proceeds in the opposite direction.) For the most part, data buffering is transparent to the application programmer. During exception conditions such as end of tape, however, you will need to decide how to handle data remaining in the buffers, as described below.

In some cases, the data you are writing will require more than one tape. When you approach the end of a tape, the SeqAccessWrite operation returns an EOM status code. This status code is the early warning signal that EOM is about to be reached. At this time, you may have unwritten data in the device's buffers (if available) and in the service's buffers. The following paragraphs describe the process of recovering this data before you change tapes and then writing the remainder of the new data.

There are two kinds of data buffers: buffers in the sequential access device and buffers in the Sequential Access Service. Use the SeqAccessModeQuery operation to determine the total size of the data buffers available, as described below. Then allocate the appropriate amount of memory for use in recovering buffer data, as described in the following paragraphs.

Device Buffers

Some devices have buffers and some do not. In addition, you can reclaim data from some device buffers but not from others. To determine the size of the device buffers, examine the *DeviceBufferSize* field returned by the SeqAccessModeQuery operation, which indicates the size in bytes of the device buffers. This size is a fixed value that cannot be changed by the programmer. In addition, check the *fDataBufRecoverable* flag, also returned by SeqAccessModeQuery. If this flag is TRUE, the Sequential Access Service can retrieve data from the device's buffers. If this flag is FALSE, the data in the device buffers is not available to the service.

Sequential Access Service Buffers

To determine the size of the Sequential Access Service buffers, examine the *ServiceBufPoolSize* field returned by the SeqAccessModeQuery operation, which indicates the size in bytes of the service buffers.

Allocating Memory for Recovering Buffer Data

To determine how much memory to allocate for recovering buffer data, add the size of the Sequential Access Service buffers to the size of the device buffers (if *fDataBufRecoverable* is TRUE). The normal sequence is to open a device, use SeqAccessModeQuery to determine the available buffer sizes, and then allocate memory for data recovery when you reach the end of a tape.

Residual Data

Each SeqAccessWrite operation requires a pointer to an area where the amount of data unsuccessfully transferred is returned (pCbResidual). The following discussion of residual data applies when an End of Medium (EOM) condition occurs. At other times, a nonzero value for residual data can be an error indicator.

When EOM is reached, you follow one of two general sequences for dealing with the residual data as described below.

- Case A: The amount of residual data is less than or equal to the size of the current write.
- Case B: The amount of residual data is greater than the size of the current write.

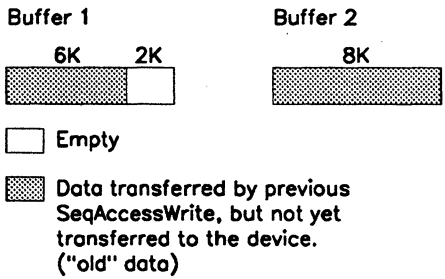
The following paragraphs outline the steps to follow for each of these two cases.

Case A is the simpler of the two cases. For purposes of this discussion, there is no need to distinguish between data in the device's buffers and data in the service's buffers. Residual data includes both kinds of buffer data. For this example, suppose the current buffer size is 4K bytes. A SeqAccessWrite operation tries to write a 4K byte block of data. The operation returns with an EOM status code and a value of 3K bytes of residual data (that is, 1K byte was successfully written to the tape). The steps to follow are given below:

1. Use the SeqAccessDiscardBufferData to discard the 3K bytes of buffered data in the device and service buffers.
2. Change the tape.
3. Since 1K byte of data in this write has already been transferred successfully, adjust the starting address of the write by 1K byte. Then reissue the SeqAccessWrite for the 3K bytes of data that have not yet been written.

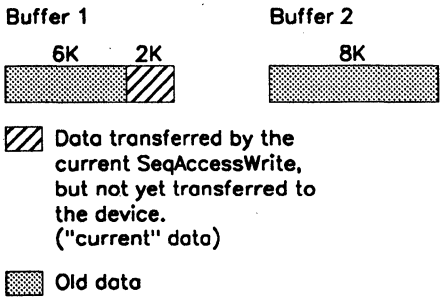
To illustrate Case B, suppose there are two 8K-byte buffers in the Sequential Access Service and no device buffers. The following events occur.

1. Buffer 1 contains 6K bytes of data, and Buffer 2 contains 8K bytes of data and is full, as shown below.



2. While attempting to write the data indicated above, the Sequential Access Service receives an EOM warning from the device. The service freezes I/O and stops writing.

3. The application issues a 4K write. Since Buffer 1 has 2K bytes empty, 2K bytes are copied into Buffer 1. Now both buffers are filled.

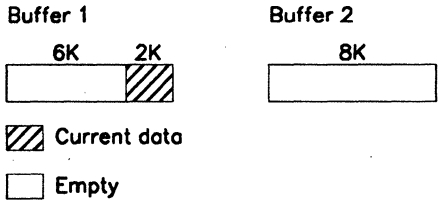


4. The SeqAccessWrite operation returns an EOM status code and a value of 18K bytes for residual data (the 14K bytes of data that were "safely" accepted before EOM was reached, the 2K bytes received after EOM, and the 2K bytes not even accepted from the user buffer).

5. When the value of residual data is greater than the size of the previous write (Case B), the amount of previously accepted data that can be recovered is equal to the difference between residual data and the size of the previous write. In this example, the values are:

   18K bytes - 4 K bytes = 14K bytes (of previously accepted data that can be recovered)

   Use the SeqAccessRecoverBufferData to recover the 14K bytes of data that was previously written to the service's buffers (step 1). This data is returned in the same order in which it was originally written (first-in, first-out) so that you can write it again (see Step 8).



6. Note in the figure above that Buffer 1 still contains the 2K bytes of data from the current write. Since there is still data in the buffers, you are not yet ready to change tapes. (A SeqAccessClose operation will fail with a status code that indicates that data is still in the buffers.)

   This 2K bytes of data is still available to the program, however, because it is part of the current write (as in Case A). So all you need to do is discard the buffered data through use of the SeqAccessDiscardBufferData operation.

7. Write the filemark.

8. The buffers are now empty, so you can change the tape.

9. Issue a SeqAccessWrite operation with the 14K bytes of data from the recovery buffer.

10. Reissue the SeqAccessWrite operation with the 4K bytes of current data. (This is the same write you were attempting when you received the EOM warning in Step 3 above.)

Buffer Recovery Order

The sequence described above assumes that data is recovered in a first-in, first-out order. To change the order to last-in, first-out, specify TRUE for the *fBufRecoverLIFO* flag in the Mode Parameters Block passed to the SeqAccessModeSet Operation.

Example

The following code fragment shows the process of recovering all unwritten data before writing on a second tape.

```
char buffer[BUFFER_SIZE];
char recovery_buffer[RECOVER_BUFFER_SIZE];
unsigned long prior_data, residual;
unsigned on_medium;
.
.
.
erc = SeqAccessWrite(handle, buffer, BUFFER_SIZE,
                     &residual);
if (erc == ercTapeEomWarning) {
   if (residual < BUFFER_SIZE) {
      on_medium - BUFFER_SIZE - residual;
      prior_data = 0;
   } else {
     on_medium = 0;
     prior_data = residual - BUFFER_SIZE;
   }
   if (prior_data != 0)
      CheckErc (SeqAccessRecoverBufferData(handle,
                recovery_buffer, prior_data, &residual));
   CheckErc(SeqAccessDiscardBufferData(handle));
   erc = SeqAccessControl(handle, CTRL_WRITE_FILEMARKS,
                          n, &residual);
   if(erc != ercTapeEomWarning)
      CheckErc(erc);
   if(residual != 0)
      ErrorExit(ercTapeOverflow);
   CheckErc(SeqAccessControl(handle, CTRL_UNLOAD,
            SYNCHRONIZE, &residual));
   /* prompt for mount of new tape */
   CheckErc(SeqAccessControl(handle, CTRL_REWIND,
            SYNCHRONIZE, &residual));
   if (prior_data != 0)
   CheckErc(SeqAccessWrite(handle, recovery_buffer,
            prior_data, &residual));
   CheckErc(SeqAccessWrite(handle, &buffer[on_medium],
            BUFFER_SIZE - on_medium, &residual));
}
.
.
```

SeqAccessCheckpoint

The SeqAccessCheckpoint operation forces all data out of the device and service buffers onto the device. This call can be used if you know that the space remaining on the medium after the early warning EOM is greater than the amount of data stored in the buffers.

Function 5, Write Filemark, of the SeqAccessControl operation contains an implicit call to SeqAccessCheckpoint, which tries to write all data onto the medium before the filemark is written.

The following code fragment shows the use of SeqAccessCheckpoint to flush all unwritten data onto the tape. This method can be used in cases where you have no buffers to hold data while the tape is being changed (for example, with tape bytestreams). In general, use the method described above in "Residual Data," if possible.

```
char buffer[BUFFER_SIZE];
unsigned long buffer_data, prior_data, residual;
unsigned on_medium;
.
.
.
erc = SeqAccessWrite(handle, buffer, BUFFER_SIZE,
                     &residual);
if (erc == ercTapeEomWarning) {
   if (SeqAccessStatus(handle, &status, sizeof(status),
       &buffer_data) == ercTapeStatusUnavailable)
      ErrorExit(ercTapeStatusUnavailable);
   if(residual < BUFFER_SIZE) {
      on_medium = BUFFER_SIZE - residual;
      prior_data = 0;
```

```
    } else {
       on_medium = 0;
       prior_data = residual - BUFFER_SIZE;
    }
    CheckErc(SeqAccessCheckpoint(handle, &residual));
    if(residual != 0)
       ErrorExit (ercTapeOverflow);
    on_medium = on_medium + (buffer_data - prior_data);
    erc = SeqAccessWrite(handle, &buffer[on_medium],
                    BUFFER_SIZE - on_medium, &residual);
    if (erc == ercTapeEomWarning || residual != 0)
       CheckErc(SeqAccessCheckpoint(handle, &residual));
    else
       CheckErc(erc);
    erc = SeqAccessControl(handle, CTRL_WRITE_FILEMARKS,
                           n, &residual);
    if (erc != ercTapeEomWarning)
       CheckErc(erc);
    if(residual != 0)
       ErrorExit(ercTapeOverflow);
    CheckErc(SeqAccessControl(handle, CTRL_UNLOAD,
             SYNCHRONIZE, &residual));
}
.
.
.
```

Specifying Buffer Sizes

Usually, the default buffer sizes will meet the needs of your application. The following guidelines are provided if you have special requirements and need to change the defaults.

- If you are operating in buffered mode with fixed-length records, each service buffer must be a multiple of the block size.
- If you are operating in buffered mode with variable-length records, the service buffer size must be greater than or equal to the maximum record size.
- A minimum of two service buffers is recommended. Increasing the number of buffers may improve performance.

Other considerations for size and number of buffers are

- Size of blocks of data transferred over the cluster to the server
- Size of the device's internal buffer and whether it is recoverable

For example, a QIC-02 device has an internal buffer capacity of 2K or 4K bytes. The current maximum transfer size over the cluster is 4K. The Sequential Access Service uses a default of four 8K-byte buffers because it fits well with the device buffer size and the cluster transfer size, and in addition, it provides a generous cushion to keep the device streaming even if the supply of data is not steady.

As another example, the SCSI QIC device has a 32K-byte internal buffer and pauses in accepting data over the SCSI bus at 16K-byte boundaries. The default buffer allocation for this device is two 32K-byte buffers. The larger size is chosen because this QIC device has a larger buffer, and this buffer size matches the natural pauses of the device.

Fixed-Length and Variable-Length Records

A record is a logical division of the physical block written on the medium. Records can be either of a fixed length or can vary in length. As described later (see "Record Size and Block Size"), some devices support only one type of record. If you specify a fixed record size, each SeqAccessWrite operation must be a multiple of this record size. If you specify variable-length records, each SeqAccessWrite operation writes a separate record, of any length. (With variable-length records, you are actually varying the physical size of each block written on the tape.)

Increasing Block Size: Pros and Cons

There are both good and bad side effects to increasing block size. In general, increasing block size increases the amount of data that can fit on the medium. Larger blocks are more efficient and faster, since larger portions of data are transferred at once. Remember, though, that a medium such as a tape contains randomly distributed defects, and increasing block size also increases the likelihood that you will run into those defects.

When the Sequential Access Service starts to write a block and runs into a defect, it erases that portion of the medium and rewrites the data on another portion. This process consumes time and wastes part of the medium. If the block size is too large, large portions of the medium may be unused.

Programming Considerations

To write your application, you need to know certain basic facts about the recording format of the sequential access device you are using before you can determine values such as minimum and maximum record size or block size. The following paragraphs indicate key considerations for the programmer.

Record Size and Block Size

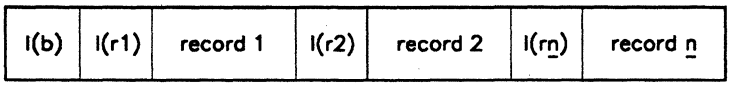
For streaming devices, such as QIC tape, record size is the determining factor, and block size is not used. Minimum record size should be set to the same value as maximum record size in the Mode Parameter Block. Many QIC devices require a record size of 512 bytes.

Some devices, such as half-inch reel-to-reel tape, allow you to group logical records into physical blocks of a fixed size. In this case, specify the same size for *MaxRecordSize* and *MinRecordSize* in the Mode Parameter Block. BlockSize, which is the size of the physical block that is written to the medium, must be a multiple of this record size.

For half-inch tape, the 4096-byte block size used in the default configuration files shipped with Standard Software (and used by the archival utilities) is chosen as a good compromise between tape speed, amount of data that can be put on the tape, and the 4K byte XBlock size commonly used on the CTOS cluster. Generally, the larger the block size, the faster the tape can be kept moving and the more data can be written on a tape.

Half-inch reel-to-reel tape and DDS also allow you to create variable-length records, in which all block sizes on the tape vary. Figure 18-5 shows one scheme for variable length records in which the first field contains the length of the block, followed by a field with the length of each record, and then the record itself. To use variable-length records, specify TRUE in the *fVariableLength* field of the Mode Parameter Block.

Figure 18-5. Example of Variable-Length Records



l(b) = length of the block
l(rn) = length of record n

606.18-5

Whether you use fixed or variable-length records depends on the application and the type of data being recorded. If the size of the fields varies widely, variable-length records may be more appropriate.

Recording Density and Transport Speed

Specifying 0 for the *Density* and/or *Speed* fields uses the default value for the particular device. You might need to specify a nondefault value if you are sending data to another system.

Buffered Mode

Currently, only buffered mode is supported. The device may or may not have buffers, but the Sequential Access Service always buffers data. (See "Data Buffering," earlier in this section.)

Erase to EOM After Close

In certain cases, perhaps for security reasons, you may need to erase the medium from the current position to the physical end of the medium to ensure that the remainder of the tape is blank. (See function 4, Erase Medium, of the SeqAccessControl operation.) Unless you have a specific need to be sure that the rest of the tape is blank, you can skip this step and save time.

A QIC tape drive has a secondary erase head that erases the tape ahead of the write head, so this step is not necessary for QIC tape. On many QIC devices, this step actually erases the data you just wrote and therefore causes serious problems.

Suppress Default Mode on Open

This option refers to SCSI devices only. By default, when a SCSI sequential access device is opened, the device is reset to the default configuration parameters. If you have a special application that needs to configure the device differently, specify TRUE for the *fSuppressDefaultOnOpen* flag in the Mode Parameters Block passed to the SeqAccessModeSet operation. This setting indicates that you do not want the Sequential Access Service to reset the device to the default parameters. Instead, the application can use SCSI Manager calls to configure the device in some special manner.

Buffer Recovery Order

The sequence described above assumes that data is recovered in a first-in, first-out order. To change the order to last-in, first-out, specify TRUE for the *fBufRecoverLIFO* flag in the Mode Parameters Block passed to the SeqAccessModeSet Operation.

Examples

Example 1: Fixed-Length, Blocked Records

The following example shows writing fixed-length, blocked records onto a sequential access device. It might be, for instance, a program to copy a payroll records file onto an unlabelled tape to be read by a mainframe system. The executive command form might look like:

```
Write Payroll Tape
    Input file
    Output tape
```

The record size is arbitrarily chosen to be 80 (most payroll systems are antiques, based in the days when data was punched on cards) and the block size is 8,000. Notice that the size of the buffers allocated by the Sequential Access Service when the tape is opened *must* be a multiple of the block size.

At the end of the file, the Sequential Access Service automatically writes a complete 8,000 characters block.

Also notice that two filemarks are written at the end of the file, both to delimit the end of this particular file and to delimit the end of recorded data on the whole tape. This reflects an assumption that the data is to be written to half-inch reel-to-reel tape. If the data were written with another recording technology, for example QIC or DDS, one filemark would suffice because those recording formats can detect blank tape (whereas half-inch cannot).

If the number of characters in the input file is not an even multiple of the record size, the application program pads the last record with garbage data before writing to the tape. A production program would probably reject this as an error.

Example 18-1. Fixed Length, Blocked Records

```
/* Standard C library macros and functions invoked by this
   module */

pragma Off(List);
#include <intel80X86.h>
pragma Pop(List);

/* Suppress C run-time (only CTOS functionality needed)*/

pragma Off(List);
#include <stub.h>
pragma Pop(List);

/* External CTOS and CTOS Toolkit functions invoked by this
   module */

#define CheckErc
#define CloseByteStream
#define Exit
#define OpenByteStream
#define ReadBsRecord
#define RgParam
#define SeqAccessClose
#define SeqAccessControl
#define SeqAccessOpen
#define SeqAccessWrite

pragma Off(List);
#include <ctoslib.h>
pragma Pop(List);

/* Type definitions used by this module */

#define BLOCK_SIZE 8000u
#define RECORD_SIZE 80u
#define MODE_MODIFY 0x6D6D
#define CTRL_REWIND 1
#define CTRL_UNLOAD 2
#define CTRL_WRITE_FILEMARK 5
#define SYNCHRONIZE 0
```

Example 18-1. Fixed Length, Blocked Records (cont.)

```
/* Error return codes used by this module */
pragma Off(List);
#include <erc.h>
pragma Pop(List);
pragma Page(1);
void main(void) {
   char bswa[130], buffer[1024], record[RECORD_SIZE];
   unsigned erc, seq_handle, transfer_count;
   unsigned long residual;
   sdType sd_device, sd_file;
   seq_parameters_type seq_parameters;
   CheckErc(RgParam(1, 0, &sd_file));
   CheckErc(OpenByteStream(bswa, sd_file.pb, sd_file.cb,
            NULL, 0, modeRead, buffer, sizeof(buffer)));
   CheckErc(RgParam(2, 0, &sd_device));
   memset(&seq_parameters, 0, sizeof(seq_parameters));
   seq_parameters.block_size = BLOCK_SIZE;
   seq_parameters.min_record_size = RECORD_SIZE;
   seq_parameters.max_record_size = RECORD_SIZE;
   seq_parameters.service_buffers = 2;
   seq_parameters.service_buffer_size = 2 * BLOCK_SIZE;
   CheckErc(SeqAccessOpen(&seq_handle, sd_device.pb,
            sd_device.cb, NULL, 0, MODE_MODIFY,
            &seq_parameters, sizeof(seq_parameters)));
   CheckErc(SeqAccessControl(seq_handle, CTRL_REWIND,
            SYNCHRONIZE, &residual));
   while ((erc = ReadBsRecord(bswa, &record,
          sizeof(record), &transfer_count)) != ercEOF) {
      CheckErc(erc);
      CheckErc(SeqAccessWrite(seq_handle, record,
               sizeof(record), &residual));
   }
   if (transfer_count != 0)
      CheckErc(SeqAccessWrite(seq_handle, record,
               sizeof(record), &residual));
   CheckErc(SeqAccessControl(seq_handle,
            CTRL_WRITE_FILEMARK, 2, &residual));
   CheckErc(SeqAccessControl(seq_handle, CTRL_UNLOAD,
            SYNCHRONIZE, &residual));
   CheckErc(CloseByteStream(bswa));
   CheckErc(SeqAccessClose(seq_handle));
   Exit();
}
```

Example 2: Variable-Length Records

The following example demonstrates writing variable-length records onto a sequential access device. To illustrate the division of an input file into records of differing lengths, a text file is parsed into lines delimited by the new-line character. It might be, for instance, a program to transfer program source code to a mainframe library system that indexes each line of a program individually for version control and retrieval. The executive command form could be:

```
Write Source Tape
     Input text file
     Output tape
```

The program assumes that no text lines in the input are longer than 256 characters; no error checking is performed.

Variable-length records might be more accurately described as variable-length blocks. The length of each physical record (or block) varies when it is written to the medium. Such variable-length blocks typically have an internal logical structure that further divides the block into individual records whose length may also vary. The structure used in this program is to have the first word of a block contain the length of the block, inclusive of the size of this descriptor field. The block length descriptor is followed by one or more self-describing records that also contain their own length in the first word.

More than one scheme for the organization of variable-length records is possible. The one chosen is common, but the important feature is that the variable-length records are self-describing. A production program, for example, would check the described length of a record read from tape with the actual data-transfer count to determine if a discrepancy exists.

Notice that *two* filemarks are written at the end of the file, both to delimit the end of this particular file and to delimit the end of recorded data on the whole tape. This reflects an assumption that the data is to be written to half-inch reel-to-reel tape. If the data were written with another recording technology, for example QIC or DDS, one filemark would suffice because devices using those recording formats can detect blank tape, whereas half-inch devices cannot.

Example 18-2. Variable Length Records

```
/* Standard C library macros and functions invoked by
 * this module
 */

pragma Off(List);
#include <intel80X86.h>
#include <string.h>
pragma Pop(List);

/* Suppress C run-time (only CTOS functionality needed)*/

pragma Off(List);
#include <stub.h>
pragma Pop(List);

/* External CTOS and CTOS Toolkit functions invoked by
 * this module
 */

#define AllocMemorySL
#define CheckErc
#define CloseByteStream
#define ErrorExit
#define Exit
#define OpenByteStream
#define ReadByte
#define RgParam
```

Example 18-2. Variable Length Records (cont.)

```
#define SeqAccessClose
#define SeqAccessControl
#define SeqAccessOpen
#define SeqAccessRead
#define SeqAccessWrite

>pragma Off(List);
#include <ctoslib.h>
pragma Pop(List);

/* Type definitions used by this module */

#define MAX_RECORD_SIZE 8192u
#define MAX_TEXT_SIZE 256u

#define MODE_MODIFY 0x6D6D

#define CTRL_REWIND 1
#define CTRL_UNLOAD 2
#define CTRL_WRITE_FILEMARK 5
#define SYNCHRONIZE 0

typedef struct {
   Boolean write_protected;
   Boolean variable_length;
   Boolean unbuffered;
   Boolean suppress_default_mode_on_open;
   unsigned speed;
   unsigned density;
   unsigned long total_blocks;
   unsigned long block_len;
   unsigned min_record_len;
   unsigned long max_record_len;
   unsigned long device_buffer_len;
   unsigned long service_buffer_pool_len;
   unsigned service_buffers;
   unsigned service_buffer_len;
   unsigned long write_buffer_threshold;
   unsigned long read_buffer_threshold;
   Boolean data_buffer_nonrecoverable;
   Boolean disable_automatic_velocity;
   Boolean buffer_recovery_LIFO;
```

Example 18-2. Variable Length Records (cont.)

```
    Boolean checkpoint_EOM;
    Boolean data_compression;
    char gap_len;
    unsigned long buffer_len_EOM;
    Boolean report_soft_errors;
    Boolean disable_error_correction;
    Boolean disable_read_retries;
    Boolean disable_write_retries;
    unsigned read_retry_limit;
    unsigned write_retry_limit;
} seq_parameters_type;

#define sdType

pragma Off(List);
#include <ctosTypes.h>
pragma Pop(List);

/* Error return codes used by this module */

pragma Off(List);
#include <erc.h>
pragma Pop(List);

pragma Page(1);

void main(void) {

    char bswa[130], buffer[1024], c, text[MAX_TEXT_SIZE],
        *variable_record;
    unsigned available = MAX_RECORD_SIZE -
         sizeof(unsigned), erc, record_len =
         sizeof(unsigned), seq_handle, text_len = 0;
    unsigned long residual;
    sdType sd_device, sd_file;
    seq_parameters_type seq_parameters;

    CheckErc(AllocMemorySL(MAX_RECORD_SIZE,
             &variable_record));
    CheckErc(RgParam(1, 0, &sd_file));
```

Example 18-2. Variable Length Records (cont.)

```
CheckErc(OpenByteStream(bswa, sd_file.pb, sd_file.cb,
        NULL, 0, modeRead, buffer, sizeof(buffer)));
CheckErc(RgParam(2, 0, &sd_device));
memset(&seq_parameters, 0, sizeof(seq_parameters));
seq_parameters.variable_length = TRUE;
seq_parameters.min_record_len = 1;
seq_parameters.max_record_len = MAX_RECORD_SIZE;
seq_parameters.service_buffers = 4;
seq_parameters.service_buffer_len = MAX_RECORD_SIZE;
CheckErc(SeqAccessOpen(&seq_handle, sd_device.pb,
        sd_device.cb, NULL, 0, MODE_MODIFY,
        &seq_parameters, sizeof(seq_parameters)));
CheckErc(SeqAccessControl(seq_handle, CTRL_REWIND,
        SYNCHRONIZE, &residual));
while ((erc = ReadByte(bswa, &c)) != ercEOF) {
   CheckErc(erc);
   if (c == '\n') {     /* Found the end of a line? */
      if (available < text_len + sizeof(unsigned)) {
         *((unsigned *) variable_record) = record_len;
         CheckErc(SeqAccessWrite(seq_handle,
                  variable_record, record_len,
                  &residual));
         available = MAX_RECORD_SIZE - (record_len =
                     sizeof(unsigned));
      }
      *((unsigned *) &variable_record[record_len]) =
        text_len;
      available -= sizeof(unsigned);
      record_len += sizeof(unsigned);
      memcpy(&variable_record[record_len], text,
             text_len);
      available -= text_len;
      record_len += text_len;
      text_len = 0;
   } else if (text_len < MAX_TEXT_SIZE)
      text[text_len++] = c;
   else
      ErrorExit(ercInconsistency);
}
```

Example 18-2. Variable Length Records (cont.)

```
    if (text_len != 0) {
       if (available < text_len + sizeof(unsigned)) {
          *((unsigned *) variable_record) = record_len;
          CheckErc(SeqAccessWrite(seq_handle,
                   variable_record, record_len,
                   &residual));
          available = MAX_RECORD_SIZE - (record_len =
                      sizeof(unsigned));
       }
       *((unsigned *) &variable_record[record_len]) =
          text_len;
       available -= sizeof(unsigned);
       record_len += sizeof(unsigned);
       memcpy(&variable_record[record_len], text,
              text_len);
       available -= text_len;
       record_len += text_len;
    }
    if (record_len > sizeof(unsigned)) {
       *((unsigned *) variable_record) = record_len;
       CheckErc(SeqAccessWrite(seq_handle,
                variable_record, record_len, &residual));
    }
    CheckErc(SeqAccessControl(seq_handle,
             CTRL_WRITE_FILEMARK, 2, &residual));
    CheckErc(SeqAccessControl(seq_handle, CTRL_REWIND,
             SYNCHRONIZE, &residual));
    CheckErc(CloseByteStream(bswa));
    CheckErc(SeqAccessClose(seq_handle));
    Exit();

}
```

Index

8274 serial controller, 9-3
8530 serial controller, 9-3
  EOF reporting, 9-8

A

a1-characters,
  CD-ROM files, 17-60
Adaptive Differential Pulse Code Modulation (ADPCM), 14-13
adding DLL client resources, 10-34
adding DLL resources, 10-26
AddQueue, 12-7
AddQueueEntry, 12-7
address space protection, 2-12
aliasing, 2-9
AllocMemoryInit, 16-32
AlphaColorEnabled, 3-27 to 3-28
alt requests, 5-4
amplifying voice messages, 14-11
analog crosspoint switch array, 14-9
Async.lib
  common-code module, 16-25
  context control block, 16-27
  contexts, 16-26, 16-28
  managing contexts, 16-25
  procedures, 16-10, 16-25
  requesting a service, 16-10
Asynchronous System Service, 16-1
  context stack, 16-15
  debugging, 16-22
  debugging statistics, 16-24
  file availability, 16-33
  linking, 16-34
  LogAsync, 16-22
  procedural interface, 16-11
  program examples, 16-36
  program flow, 16-7
  request blocks, 16-14
  requests, 16-10 to 16-11
  source modules, 16-9
  synchronous and, 16-3
  terminology, 16-1
  variable length parameter list, 16-12
  writing, 16-6
Asynchronous Terminal Emulator (ATE), 14-2
AsyncRequest, 16-14
AsyncRequestDirect, 16-14
attribute byte, 3-24, 4-3
attribute map, 4-23
audio features,
  of CD-ROM, 17-24
Audio Pause, 17-25
audio play, 17-24
  example of, 17-25
Audio Q-Channel Info function, of CD-ROM Service, 17-24
Audio Resume, 17-25
Audio Status, 17-25
automatic data segment, 10-18 (See also DGroup)
automating DLL building, 10-22 to 10-26
awk, 1-1
a-characters,
  CD-ROM files, 17-59

B

background color, 3-2, 4-3
backslash, in CD-ROM file specification, 17-8
batch, 6-6, 6-23
Batch.run, 6-1
baud rate, 9-4
blank space, on tape, 18-7
block size, 18-17
blocks, increasing size of, 18-17
buffers
  determining size of, 18-8
  recovery order, 18-12, 18-19
  specifying size of, 18-16
BuildAsyncRequest, 16-11
BuildAsyncRequestDirect, 16-12

C

call gate, 2-12
call progress tone detection (CPTD), 14-9, 14-18
CD-ROM disc
  directory list buffer, 17-9
  flat file structure, 17-6
  hierarchical file structure, 17-6
  primary volume descriptor, 17-4
CdAbsoluteRead, 17-4
CdAudioCtl, 17-3, 17-24
  example, 17-25
CdClose, 17-3
CdControl, 17-3
CdDirectoryList, 17-2, 17-9 to 17-10
  example of, 17-11
CdGetDirEntry, 17-2, 17-19, 17-49
CdGetVolumeInfo, 17-2, 17-4
CdOpen, 17-3
CdRead, 17-3
CdSearchClose, 17-2, 17-13
CdSearchFirst, 17-2, 17-13
CdSearchNext, 17-2, 17-13
CdServiceControl, 17-4
CdVerifyPath, 17-3
CdVersion, 17-3
CdVersionRequest, 17-13
CD-ROM
  character sets, 17-58
  example of file search, 17-13 to 17-14
  file, copying to disc, 17-19
  file formats, 17-33
  High Sierra directory record format, 7-54
  High Sierra primary volume descriptor, 7-39
  ISO directory record format, 17-49
  ISO primary volume descriptor, 17-33
  searching for files, 17-1
CD-ROM disc
  file structures used, 17-6
  specifying locations on, 17-24
CD-ROM Service
  audio features, 17-24
  file formats, 17-4
  function of, 17-1
  installation,17-1
  operations, 17-2
  requirements for, 17-1
ChangeCommLineBaudRate, 9-4
character
  attribute map, 4-23
character attribute byte, 3-24
character attributes, 4-17
character cell, 3-2, 4-2
character code, 4-3
character color, 3-25

character map, 3-3, 4-22
  accessing, 4-40
character sets
  for CD-ROM, 17-58
CheckContextStack, 16-15
child partition termination status, 5-6
circular queue, creating, 10-7 to 10-8
CleanQueue, 12-8
cleanup procedures (See DLL termination procedures)
Clients.mak, 10-55 to 10-57
client-server model, 1-1
clock source, 9-5
cluster network, 1-1
cluster server, 18-1
CODEC (encoder/decoder), 14-3, 14-6, 14-10, 14-23, 14-27
color intensity
  three-palette format, 3-10
color intensity, 3-7, 3-9
color programming, 3-1
  and graphics, 3-5
  character attribute byte, 3-24
  color priority, 3-28
  graphics colors, 3-27
  palette control structure, 3-11
  single-palette format, 3-7
  three-palette color format, 3-9
command file editor, 6-6, 6-14
command file, for installation, 6-6, 6-14
CommLine interface, 9-1
  baud Rate, 9-4
  clock source, 9-5
  extensions, 9-1
  reading signal status, 9-4
  serial controller differences, 9-3
  setting signal status, 9-5
  using DMA with, 9-5
communication controller, 9-3
communications DMA, 9-5
  and DTR signal, 9-5
  and WriteCommLineStatus, 9-5
  external/Status interrupt, 9-7
  getting status, 9-8
  initializing, 9-6
  receive special interrupt, 9-8
  receiving data, 9-7
  transmitting data, 9-7
communications line configuration block, 9-2
  fDMA field, 9-6
  fV35Mode field, 9-13
  fX21 field, 9-12
communications line return area, 9-2
  dMAAvailable field, 9-6
  fV35Avail field, 9-13
  ioX21 field, 9-11
compiler options, CTOS Microsoft C 10-1
compiling and building DLLs, 10-19 to 10-26
compiling DLL clients, 10-31
compiling DLLs, 10-23
compressing pauses
  in voice files, 14-10
Config.sys, 3-14
  parameters, 1-17
configuration
  spooler, 13-1
ConfigureSpooler, 13-2
connection handle, 8-20
context control block, 16-27
Context Manager, 5-1, 5-6, 5-8, 6-15, 6-24, 14-3

contexts, 16-26
  control block, 16-27
  creating, 16-28
  terminating, 16-29
  uses, 16-28
control file, for installation, 6-6 to 6-7
converting CTOS VGA, 4-1
  concepts and terminology, 4-2
  mapping character attributes, 4-23
  VAM 4.1.5, 4-1
copying a CD-ROM file to disc, example of, 17-19
CPTD configuration file, 14-19
Create Message File command, 6-13
CreateContext, 16-26
creating DLL import libraries, 10-24
creating DLL object modules, 10-24
creating dynamic link libraries, 10-1 to 10-57
creating partitions, 5-2
CTOS, 1-1
  calling convention, 1-18
  development tools, 1-2
  model of segmentation used, 1-17
  naming conventions, 10-6
  protection model used, 2-13
  system calls, 1-5
  system debugger, 1-19
  system software, 1-2
  use of call gates, 2-12
  use of GDT-based segments, 2-8
CTOS character attributes, 4-20
CTOS III, 10-1
CTOS naming conventions, 10-6
CTOS semaphore functions, 10-14
CTOS VGA
  converting to standard VGA, 4-1
  VAM 4.1.5, 4-1
  VAM default colors, 4-25
  default colors, 4-21
CTOS.lib version consistency, 1-20
CTOS/XE, 8-1
  exchanges and user numbers, 8-2
  standard connection handle, 8-21
CtosLib.dll, 10-2
CtosLibimp.lib, 10-3
cursor
  graphics, 11-14
  movement, 11-15
  shape, 11-12
cyclic redundancy check (CRC), 9-7
c-characters,
  CD-ROM files, 17-60

D

d1-characters,
  CD-ROM files, 17-60
data
  global, 10-6
  instance 10-6
  nonshared (See instance data)
  reading and writing, 14-24
  shared (See global data)
data and voice
  separate lines for, 14-15
data blocks
  on tape, 18-6
data call
  accepting, 14-23
  converting a voice call to, 14-23
  example, 14-22
  originating, 14-24
  starting, 14-23
  terminating, 14-24

data control structure, 14-23
data management, 4-2, 14-2
data statement, 10-18
data storage characteristics
  of sequential access devices, 18-4
data terminal ready (DTR), 9-5
data types, 1-9
  defining, 1-10
data, global 10-6
DDS devices, 18-18
  recording data on, 18-6
debugger, 1-19
default attribute mapping tables, 4-29
  monochrome, 4-30 to 4-31
  with alphanumeric color, 4-32 to 4-39
  without alphnumeric color, 4-30
default character attribute mapping
  standard VGA, 4-24
default colors
  CTOS VGA, 4-21
  standard VGA, 4-19
default system font, 4-10
defining DLL default data types, 10-18
defining DLL instance data, 10-18
DeinstallQueueManager, 12-8
dependents, 10-23
descriptor table, 2-4
development library consistency, 1-20
device routing, 8-4
devices, supported by Sequential Access Service, 18-1
DGroup, 10-18
dialer, 14-28
dialing telephone numbers, 14-16
digital data storage (DDS), 18-1
digitizing voice, 4-3
directory list buffer
  for CD-ROM disc, 17-9
directory list
  for CD-ROM disc, 17-9
disk activity, 5-5, 15-1
disk requirements
  for voice files, 14-12
disk seeks, 14-12
display resolution, 4-15
distributed computing, 1-1
DLL
  source files, 10-5
  client, 10-4, 10-27
  initialization procedures, 10-11 to 10-12
  initialization types, 10-11
  instance data, defining 10-18
  termination procedures, 10-12 to 10-16
DLL.mak, 10-47 to 10-48
DLL_Term.obj, 10-3
DMA for communications (See communications DMA)
DTMF
  encoder, 14-3
  generator and receiver, 14-3, 14-7
  generating tones, 14-17 to 14-18
dynamic link libraries
  creating, 10-1 to 10-57
dynamic linking
  loadtime 10-2
  runtime 10-2
d-characters,
  CD-ROM files, 17-58

E

early warning (EW), of sequential access devices, 18-4
edf files, 1-20
editor, 1-6
end of data frame (EOF), 9-8
End of Medium condition, 18-9
enhanced video, 3-2
EnterDebuggerOnFault, 1-17
Environment Service, 10-3, 10-27
erase gaps, on tape, 18-7
error checking, 10-5
EstablishQueueServer, 12-8
exception, 2-11
executive, 1-1, 3-4, 6-6
exit procedures (See DLL termination procedures)
ExitListSet, 10-12 to 10-13
export procedures, 10-10, 10-11
external/Status interrupt, 9-7, 9-11

F

fault, 2-11
fBackgroundColor, 3-13, 3-16, 3-28
fDataBufRecoverable, 18-8
FIFO, 10-4
file formats
  for CD-ROM, 17-4, 17-33
file handle, 8-20
file suffix conventions,
file system activity, 15-1
filemarks, 18-20
  on tape, 18-7
fixed-length records, 18-16
flat file structure
  CD-ROM disc, 17-6
floppy installation, 6-2
  naming files, 6-16
font, 4-3
foreground color, 3-2, 4-3
fSuppressDefaultOnOpen, 18-19

G

gaps, on tape, 18-7
gate descriptor, 2-12
general protection fault, 2-11
GetCommLineDMAStatus, 9-8
GetQmStatus, 12-8
GetWsUserName operation, 8-3
global data variables, 10-17
global descriptor table, 2-8
global DLL initialization, 10-11
graphics and color, 3-5
GraphicsColorEnabled, 3-29
GraphicsEnabled, 3-29
gray-scale monitors, 3-23

H

half-bit shift, 4-12
half-inch reel-to-reel devices, 18-1, 18-18
  recording data on, 18-6
handle, 8-20, 8-21
header files, 1-20
heap memory, 16-15
  allocating, 16-16
  conserving, 16-17
  deallocating, 16-16
  debugging, , 16-32
  initializing, 16-32
  managing, 16-30
  using, 16-29

HeapAlloc, 16-16, 16-31
HeapFree, 16-16, 16-31
HeapInit, 16-30
helical scan recording, 18-6
hierarchical file structure
  for CD-ROM disc, 17-6
High Sierra directory record format, 7-54
High Sierra primary volume descriptor
  for CD-ROM, 7-39
High Sierra standard for CD-ROM, 17-1
hold
  placing a telephone line on, 14-16
horizontal scan line, 4-3

I

InitCommLine, 9-2
  and DMA, 9-6
  selecting extended features, 9-2
  V.35 protocol support, 9-13
  X.21 protocol support, 9-12
initialization, DLL, 10-11
Install Sequential Access Service command, 18-1
installation database, 6-4
Installation Manager
  file lists created by, 6-27
  verify feature, 6-10
installation media, 6-2
  organizing, 6-20
installation scripts
  examples, 6-34
  file, 6-6, 6-13
  tips for creating, 6-30
installation variables, 6-4, 6-23
installation, restarting, 6-28
instance data, 10-6
instance DLL initialization, 10-11
Intel documentation titles, 2-1
intercharacter space, 4-4
internationalized call progress tone detection, 14-18
  example of, 14-21
interrupt service routine
  and DMA, 9-5
  and External/Status interrupt, 9-7
  and receive special interrupt, 9-8
inter-CPU communication, 8-3
ISO directory record format, for CD-ROM, 17-49
ISO primary volume descriptor, for CD-ROM, 17-33
ISO-9660 standard, for CD-ROM, 17-1

J

JCL files, 6-1, 6-13
  examples, 6-34

K

kernel, 1-3

L

LIBRARY statement, 10-11
library version consistency, 1-20
Link command, 1-7
Link V6 command, 1-6, 17-10
linker configuration file, 10-3, 10-27 (See also *LinkerConfig.sys*)
linker, 1-6
LinkerConfig.sys, 10-3, 10-26, 10-27, 10-33, 10-57
linking DLL clients, 10-32 to 10-33
linking DLLs, 10-25
list file, 1-7
LLibCDll.lib, 10-3
LoadInteractiveTask operation, 5-6
loadtime linking, 10-2
local descriptor table, 2-7
local routing, 8-5, 8-18
locking semaphores, 10-15
log file, 7-1
  and volume home block, 7-3, 7-6
  chronological order, 7-7
  for installation, 6-10
  format of, 7-2
  reading, 7-6
  record header and trailer, 7-2
  wraparound, 7-5 to 7-6
  writing records to, 7-2
  written by file system, 7-4
LogAsync, 16-22
logging session
  closing, 15-10
  for Performance Statistics Service, 15-1, 15-8
  opening, 15-8
  reading a log, 15-9
logical address, 2-2
LogMsgIn, 16-22, 16-32
LogRequest, 16-22, 16-32
LogRespond, 16-22, 16-32

M

make, 1-1
makefiles, using 10-22 to 10-26
map file, 1-8
mapping character attributes, 4-23
mapping CTOS attributes, 4-41
MarkKeyedQueueEntry, 12-8
MarkNextQueueEntry, 12-8
MC0_Dll.obj, 10-3
mediating medium-model procedures, 10-10
memory addressing, 2-2
Merge Command Files command, 6-14
message file, for installation, 6-6, 6-13
minute-second-frame (MSF) format, on CD-ROM disc, 17-24
mixing programming languages, 1-18
models of segmentation, 1-17
modem, 14-1 to 4-2, 14-5, 14-9, 14-28
  asynchronous use of, 14-24
Module Definition command, 10-24
module definition file, 10-6, 10-11, 10-18
monochrome graphics, 3-24
motion rectangle, 11-7

Mouse Services, 6-7, 6-14, 11-1
  important concecpts, 11-3
  mouse buttons, 11-4
  procedures, 11-2
  programming examples, 11-3
  queries, 11-7
  screen coordinates, 11-4, 11-5
  setup, 11-6
  troubleshooting, 11-16
multiple voice messages
  in one file, 14-14

N

naming conventions, 1-9, 10-6
naming floppy installation files, 6-16
naming tape installation files, 6-18
nationalization, 6-13, 6-29
network, 14-2
nine-bit font capabilities, 4-6
NMake command, 10-3, 10-27
nonshared data (See DLL instance data)
NotifyCM request, 5-6
NULL pointer, 2-11

O

object file, 1-7
object module library version consistency, 1-20
object module library, 11-1
object module procedure, 1-3
obtaining semaphores, 10-15
offhook, 14-16
operating system calls, 1-3
operations for updating the screen, 4-42
operator, 14-2
overriding DLL defaults, 10-18

P

package, 6-4
paging, 2-6
palette
  alphanumeric vs. graphics, priority, 3-28
palette control structure, 3-11
  ProgramColorMapper
palettes, 3-6
  sample, 3-17
paragraph, 2-3
parallel recording, 18-6
parity, 14-5
partition management, 5-1
  Action+Finish and swapping, 5-7
  and child termination, 5-6
  sample program, 5-10
  termination procedure, 5-8
  use of termination requests, 5-7
partition type, 5-3
partitions
  consequences of unsuccessful task load, 5-6
  creating, 5-2
  deallocating, 5-3
  deallocating, 5-9
  initializing, 5-4
  loading a task into, 5-6
password
  spooler, 13-2
pause compression
  advantages and disadvantages of, 14-11
pause compression, 14-10
PBX systems, 14-7, 14-16
PDLoadCursor, 11-14
PDReadIconFile, 11-14
PDSetCursorDisplay, 11-14
PDSetCursorPos, 11-7

PDSetCursorType, 11-6
PDSetTracking, 11-7
PDSetVirtualCoordinates, 11-6
PDTranslateNSCtoVC, 11-5
PDTranslateVCtoNSC, 11-5
Pentium, 2-3 to 2-4
Performance Statistics Service
  example of, 15-10
  function of, 15-1
performance statistics structure, 15-4, 15-17
pixel, 3-2
playback, of compressed voice files, 14-11
PLog utility, 7-1, 7-3
  record-processing algorithm, 7-10
porting to protected mode, 8-1
preparing DLL clients, 10-29 to 10-30
preparing DLLs, overview, 10-20 to 10-21
primary volume descriptor
  for CD-ROM disc, 17-4
private installation, 6-3
procedure naming, 1-14
procedures
  DLL initialization, 10-12
  DLL termination, 10-12
  ExitListSet, 10-12 to 10-13
  export, 10-11, 10-16
  setting priorities of, 10-13
processor activity, 15-1
ProgramColorMapper, 3-1
  and monochrome graphics, 3-24
  control structure, 3-6
  functions performed by, 3-5
  single-palette format, 3-7
  three-palette format, 3-9
programming languages, 1-2
protected mode
  address calculation in, 2-4
  descriptor tables, 2-7
  exceptions and faults, 2-11
  features of, 2-2
  introduction to, 2-1
protected mode run file, 1-6
PsCloseSession, 15-2, 15-10
PsGetCounters, 15-2, 15-3, 15-6
PsOpenLogSession, 15-2
PsOpenStatSession, 15-3, 15-4
PsReadLog, 15-2, 15-9
PsResetCounters, 15-2
public installation, 6-3, 6-10, 6-26

Q

QIC tape device, 18-16
  recording data on, 18-5
quarter-inch cartridge (QIC) tape, 18-1
QueryVideo, 3-14, 3-23
queue entries, 12-4
  adding, 12-6
  calculating size, 12-13
  deleting, 12-7
  description, 12-12
  format, 12-13
  handles, 12-18
  header, 12-27
  keys, 12-19
  marking, 12-4
  processing order, 12-12
  processing, 12-5
  queue management, 13-5
  reading, 12-7

referencing, 12-18
removing, 12-5
rescheduling, 12-5
rewriting, 12-5
spooler, 13-5
status blocks, 12-19
unmarking, 12-5
Queue Manager, 12-1
client operations, 12-6
configuration, 12-10
data structures, 12-24
deinstallation, 12-2
installation, 12-2
network nodes, 12-22
operations, 12-3
run files, 12-2
sequence, 12-21
summary of operations, 12-7
queue server, 12-4
queue, 10-4
Queue.c, 10-35 to 10-46
Queue.def, 10-6, 10-18, 10-46
Queue.h, 10-49
queues
defining dynamically, 12-16
defining, 12-14
examples, 12-14
file header, 12-11, 12-24
format, 12-11
handles, 12-18
index file, 12-14
names, 12-17
referencing, 12-17
status block, 12-29
Q-channel, 17-24

R

ReadCommLineStatus, 9-4
ReadKeyedQueueEntry, 12-8
ReadNextQueueEntry, 12-8
ReadQueue.c, 10-50
real mode address calculation, 2-3
receive special interrupt, 9-8
ReceiveCommLineDma, 9-7
record size, 18-17
record/playback
typical sequence for, 14-14
recording data
on half-inch tape devices, 18-6
on QIC tape devices, 18-5
recording densities, 18-4, 18-18
recording rates
for voice, 14-10
recording voice, 14-10
records, Sequential Access Service, 18-16
recovering buffer data, 18-9, 18-13
Red Book standard, 17-24
remote processor memory, 8-3
remote routing, 8-4, 8-15
RemoveKeyedQueueEntry, 12-9
RemoveMarkedQueueEntry, 12-9
RemoveQueue, 12-9
request blocks, 16-14
request routing, 8-3
across the cluster, 1-2
inter-board routing directives, 8-4
request, 1-3

requests
  aborting, 16-17
  handling, 16-19
  swapping, 16-18
  terminating, 16-17
RescheduleMarkedQueueEntry, 12-9
ResetVideoGraphics, 3-14, 3-23
residual data, 18-9 to 18-10
Resource Librarian command, 10-26, 10-34
restarting an installation, 6-28
restartLabel, 6-28
Restore command, 17-10
ResumeContext, 16-27
RewriteMarkedQueueEntry, 12-9
routing by device specification, 8-6
run command, 1-8
run file mode, 1-6, 2-8
run file, 1-7
runtime linking, 10-2

S

SCSI devices, 18-19
SCSI interfaces, 18-1
Searching
  for CD-ROM files, 17-13
segment address, 2-4
segment descriptor format, 2-9
segmented addressing, 2-2
  in protected mode, 2-5
segments statement, 10-6, 10-18
selector, 2-4
  format of, 2-7
semaphore functions, CTOS 10-14
SemClear, 10-16
SemLock, 10-14, 10-16
SemOpen, 10-14
Separators,
  CD-ROM files, 17-60
SeqAccessCheckpoint, 18-3, 18-14
SeqAccessClose, 18-2, 18-12
SeqAccessControl, 18-2, 18-14
SeqAccessDiscardBufferData, 18-3, 18-10
SeqAccessModeQuery, 18-3, 18-8
SeqAccessModeSet, 18-3, 18-12
SeqAccessOpen, 18-2
SeqAccessRead, 18-2
SeqAccessRecoverBufferData, 18-3
SeqAccessStatus, 18-2
SeqAccessVersion, 18-3
SeqAccessWrite, 18-2, 18-9
sequential access devices
  data storage characteristics of, 18-4
  model of, 18-4
Sequential Access Service
  data buffering, 18-8
  determining size of service buffers, 18-8, 18-9
  devices supported by, 18-1
  function of, 18-1
  installing multiple instances, 18-1
  placement of in cluster, 18-1
  programming considerations, 18-17
  records, 18-16
  recovering buffer data, 18-9
SequentialAccessControl, 18-7
serial controller, 9-3
  and DMA, 9-6
  CTS signal and X.21, 9-10
  external/Status interrupt, 9-7
  initializing for X.21, 9-12
  receive special interrupt, 9-8

serpentine recording, 18-5
server installation, 6-2
shared data (See global data)
shared resource processor (SRP), 8-3, 18-1
  and GetWsUserName, 8-3
  device specification, 8-6
  exchanges on, 8-2
  inter-CPU communication on, 8-3
  multi-instance system services, 8-6
  porting real-mode applications, 8-1
  programming guidelines, 8-2
  request routing, 8-3
  sample *request.txt* file for, 8-7, 8-15
  special handle types, 8-21
  use of handles, 8-20
  user numbers on, 8-2
simple guidelines for
  creating makefiles, 10-22
single-palette color format, 3-7
  advantages of, 3-9
source code files, 1-6
spawn, 5-1
spooler
  configuration, 13-1
  control queue, 13-6, 13-14
  data structures, 13-8
  escape sequences, 13-5
  file requirements, 13-4
  introduction, 13-1
  operations, 13-2
  password, sending, 13-2
  pre-GPS, 13-3
  program examples, 13-7
  programmer's notes, 13-3
  queue management, 13-5
  queue scheduling, 13-6
  scheduling queue, 13-8
  status queue, 13-6, 13-11
SpoolerPassword, 13-2
standard connection handle, 8-21
standard VGA
  character attributes, 4-26 to 4-29
  default character attribute mapping, 4-24
  default colors, 4-19
statistics ID block, 15-3, 15-4
statistics session,
  closing, 15-8
  for Performance Statistics Service, 15-1
  opening, 15-4
subpackages, 6-3
SwapContextUser, 16-21
SwapInContext, 5-2
symbol file, 1-8
synchronous data communication, 9-1
  baud Rate, 9-4
  clock source, 9-5
  extensions to traditional interface, 9-1
  reading signal status, 9-4
  selecting extended features, 9-2
  serial controller differences, 9-3
  setting signal status, 9-5
  using DMA, 9-5
  V.35 protocol support, 9-13
  X.21 protocol support, 9-9
synchronous system service
  and asynchronous, 16-3
  model program flow, 16-5
  procedural interface, 16-10
syntax, makefiles 10-22 to 10-26
system configuration tips, 1-19

system log file, 7-1
  and volume home block, 7-3, 7-6
  chronological order, 7-7
  format of, 7-2
  reading, 7-6
  record header and trailer, 7-2
  wraparound, 7-5 to 7-6
  writing records to, 7-2
  written by file system, 7-4
system service, 8-6
System.dll, 10-2
Systemimp.lib, 10-3
system-common procedure, 1-3
  and the GDT, 2-8

T

tape
  logical elements within, 18-7
tape installation, 6-2
  naming files, 6-18
  organizing files, 6-23
tape, general layout of, 18-5
targets, 10-23
Task Management Service, 10-3, 10-27
telephone lines, 14-7
telephone management, 14-2
telephone Service, 4-1
  data call example, 14-69
  dialing example, 14-30
  voice memory playback example, 14-57
  voice response system example, 14-36
telephone service configuration block, 14-4
TerminateContextUser, 16-19
TerminateQueueServer, 12-9
termination requests, 5-7, 5-18
three-palette color format, 3-9
track number, on CD-ROM disc, 17-24
track, definition of, 18-5
TransmitCommLineDma, 9-7
transport speed, 18-18
troubleshooting, 1-20
TsConnect operation, 14-3, 14-3
TsDataChangeParams operation, 14-5
TsDataCheckpoint operation, 14-5
TsDataClose operation, 14-23
TsDataCloseLine operation, 14-5
TsDataOpenLine operation, 14-5, 14-23, 14-24
TsDataRead operation, 14-5, 14-23, 14-24
TsDataRetrieveParams operation, 14-5
TsDataUnacceptCall operation, 14-5
TsDataWrite operation, 14-5, 14-23, 14-24
Telephone Status command
  function keys, 14-29
  screen, 14-26
Telephone Status command, 14-1 to 14-2, 14-25
telephone status monitor program, 14-2
telephone status structure, 4-4
telephone unit
  parts of, 4-9, 14-7, 14-28
  versus telephone line, 14-16

TerminateContext, 16-28
TsDeinstall operation, 14-3
TsDial operation, 14-3, 14-17
TsDoFunction operation, 14-3, 14-16
TsGetStatus operation, 14-4
TsHold operation, 14-4, 14-16
TsLoadCallProgressTones operation, 14-4
TsOffHook operation, 14-4, 14-16
TsOnHook operation, 14-4, 14-16
TsQueryConfigParams operation, 14-4
TsReadTouchTone operation, 14-4
TsRing operation, 14-4
TsSetConfigParams operation, 14-4
TsVersion operation, 14-4
TsVoiceConnect operation, 14-3
TsVoicePlaybackFromFile operation, 14-3, 14-15
TsVoiceRecordToFile operation, 14-3
TsVoiceStop operation, 14-3

U

UnmarkQueueEntry, 12-9
user configuration file, 6-15
user number, 8-2
using DLLs, 10-27
using macros, 10-22
using makefiles, 10-22 to 10-26

V

V.35 protocol support hardware, 9-13
variable length parameter list, 16-12
variable names, 1-10
  examples, 1-13
  prefixes, 1-10
  roots, 1-12
  suffixes, 1-13
variable-length records, 18-16, 18-18, 18-25
VGA Character Attribute Mapper, 4-23
VGA, 3-2, 3-14
Video Graphics Array (See VGA)
video resource file, 4-23
video, during installation, 6-15
virtual address, 2-5
virtual screen coordinates, 11-4
voice amplifier, 14-6
voice and data
  separate lines for, 14-15
voice control structure, 14-3, 14-10, 14-14
voice file
  structure of a, 14-13
voice file header, 14-13
voice file record, 14-13
voice playback
  from memory, 14-15

voice processor module, 14-1
  connections, 4-10
  data features of, 14-9
  voice features of, 14-6
voice recording, 14-10
voice response system, 14-8
Voice/Data Services
  debugging using Telephone Status command, 14-25
Voice/Data Services
  definition of, 14-1
  functional operation groups, 14-2
  hardware features used by, 14-6
volume home block
  and log file, 7-3, 7-6

W

work area
  for the Telephone Service, 14-12
WriteCommLineStatus, 9-5
WriteQueue.c, 10-51 to 10-54
writing filemarks on tape, 18-7

X

X-Bus workstations, 14-1
X.21 protocol
  general description, 9-9
  signal lines used, 9-9
  special signaling bit patterns, 9-10
X.21 protocol support, 9-9
  and external/status interrupt, 9-11
  hardware features, 9-10
  initializing communications with, 9-12
  use of CTS signal, 9-10
X.21 protocol support hardware
  drivers-only mode, 9-12
  enabling and disabling, 9-12
  features of, 9-10
XE-530, 8-1
XmitCommLineDMA, 9-7

Y

yacc, 1-1

45867918-000